André Filipe
Pereira Alves

DICOOGLE: No-SQL para Suporte de Ambientes Big Data

DICOOGLE: No-SQL for Supporting Big Data Environments

Universidade de Aveiro
Departamento de Eletrónica,
Telecomunicações e Informática
2016

André Filipe Pereira Alves

DICOOGLE: No-SQL para Suporte de Ambientes Big Data

DICOOGLE: No-SQL for Supporting Big Data Environments

"*Challenges are what make life interesting and overcoming them is what makes life meaningful.*"

— Joshua J. Marine

Universidade de Aveiro
2016

Departamento de Eletrónica,
Telecomunicações e Informática

André Filipe Pereira Alves

DICOOGLE: No-SQL para Suporte de Ambientes Big Data

DICOOGLE: No-SQL for Supporting Big Data Environments

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Carlos Costa, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais, irmão e amigos pelo incansável apoio.

o júri / the jury

presidente / president

Prof. Doutor Augusto Marques Ferreira da Silva

Prof. Auxiliar do Dep. Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Joel P. Arrais

Prof. Auxiliar Convidado Dep. Informática da Universidade de Coimbra

Prof. Doutor Carlos Manuel Azevedo Costa

Prof. Auxiliar do Dep. Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos /
acknowledgements

Agradeço especialmente ao professor Carlos Costa e ao Tiago Godinho, pelo que me ensinaram, por estarem sempre prontos a ajudar e por me ajudarem a crescer a nível profissional.

Palavras Chave

Dados, Repositórios, Indexação, Medicina, Relacional, Não-Relacional, PACS, DICOM, Imagiologia Médica, Big Data, Optimização de Performance

Resumo

Os últimos anos têm sido caracterizados por uma proliferação de diversos tipos de modalidades de imagem médica nas instituições de saúde. Por outro lado, assistimos a uma migração de serviços para infraestruturas na Cloud. Assim, para além de um cenário onde são produzidos tremendos volumes de dados, caminhamos para uma realidade em que os processos são cada vez mais distribuídos. Tal realidade tem colocado novos desafios tecnológicos ao nível do arquivo, transmissão e visualização, muito particularmente nos aspetos de desempenho e escalabilidade dos sistemas de informação que lidam com a imagem. O grupo de bioinformática da universidade de Aveiro tem vindo a desenvolver um inovador sistema distribuído de arquivo de imagem médica, o Dicoogle Open Source PACS. Este sistema substituiu o tradicional motor de base de dados relacional por um mecanismo ágil de indexação e recuperação de dados. Desta forma é possível extrair, indexar e armazenar todos os metadados das imagens, incluindo eventuais elementos privados, sem necessidade de processos de reengenharia ou reconfiguração. Entre outros casos de uso, este sistema já indexou mais de 22 milhões de imagens em 3 hospitais da região de Aveiro. Atualmente, o Dicoogle dispõe de uma solução baseada na biblioteca Apache Lucene. No entanto, esta tem demonstrado alguns problemas de desempenho em ambientes em que temos necessidade de manusear e pesquisar sobre uma grande quantidade de dados, muito particularmente em cenários de análise de dados. No âmbito desta dissertação foram estudadas diferentes tecnologias capazes de suportar uma base dados de um repositório de imagem. Em sequência, foram implementadas quatro soluções baseadas em bases de dados relacionais, NoSQL e motor de indexação. Foi também desenvolvida uma plataforma de testes de desempenho e escalabilidade que permitiu efetuar uma análise comparativa das soluções implementadas. No final, é proposta uma arquitetura híbrida de base de dados de imagem médica que foi implementada e validada. Tal proposta demonstrou ter ganhos significativos ao nível dos tempos de pesquisa de conteúdos e em cenários de análise alargada de dados.

Keywords

Data, Archive, Indexing, Medicine, Relational, NoSQL, PACS, DICOM, Medical Imaging, Big Data, Performance Optimization

Abstract

The last few years have been characterized by a proliferation of different types of medical imaging modalities in healthcare institutions. As a result, the services are migrating to infrastructures in the Cloud. Thus, in addition to a scenario where tremendous amounts of data are produced, we walked to a reality where processes are increasingly distributed. Consequently, this reality has created new technological challenges regarding storage, management and handling of this data, in order to guarantee high availability and performance of the information systems, dealing with the images. An Open Source Picture Archive and Communication System (PACS) has been developed by the bioinformatics research group at the University of Aveiro labeled Dicoogle. This system replaced the traditional relational database engine for an agile mechanism, which indexes and retrieves data. Thus it is possible to extract, index and store all the image's metadata, including any private information, without re-engineering or reconfiguration process. Among other use cases, this system has already indexed more than 22 million images in 3 hospitals from the region of Aveiro. Currently, Dicoogle provides a solution based on the Apache Lucene library. However, it has performance issues in environments where we need to handle and search over large amounts of data, more particularly in data analytics scenarios. In the context of this work, different technologies capable of supporting a database of an image repository were studied. In sequence, four solutions were fully implemented based on relational databases, NoSQL and two distinct text engines. A test platform was also developed to evaluate the performance and scalability of these solutions, which allowed a comparative analysis of them. In the end, it is proposed a hybrid architecture of medical image database, which was implemented and validated. This proposal has demonstrated significant gains in terms of query, index time and in scenarios where it is required a wide data analyze.

Contents

List of Figures

List of Tables

Acronyms

DICOM	Digital Image Communications in Medicine
PACS	Picture Archive and Communication System
IT	Information Technology
HIS	Hospital Information System
RIS	Radiology Information System
NEMA	National Electrical Manufacturers Association
ACR	American College of Radiology
TLV	Tag Length Value
UID	Unique Identifier
WADO	Web Access to DICOM persistent
SOP	Service Object Pair
SCP	Service Class Provider
SCU	Service Client User
XML	Extensible Markup Language
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
DBMS	Database Management System
RDBMS	Relational Database Management System
SQL	Structured Query Language
NoSQL	Non Structured Query Language
RDBMS	Relational Database Management System
P2P	Peer-to-Peer
DIM	DICOM Information Model
SDK	Software Development Kit
IEETA	Instituto de Engenharia Electrónica e Telemática de Aveiro
DIMSE	DICOM Message Service Element
IOD	Information Object Definition
MR	Magnetic Resonance
JDBC	Java Database Connectivity
BSON	Binary JSON
CT	Computed Tomography
CR	X-Rays
US	Ultrasounds
ACID	Atomicity, Consistency, Isolation, Durability
RAM	Random Access Memory
CSV	Comma-separated values
API	Application Programming Interface
URI	Uniform Resource Identifier
ORM	Object/Relational Mapping
ODBC	Open Database Connectivity

CHAPTER 1

INTRODUCTION

This chapter provides an introduction and a slight overview of the main issues related to the medical imaging, some important concepts of this technique and the Dicoogle PACS platform will be introduced. Next, the main goals behind this thesis work will be presented followed by the outlines containing the most important points discussed in each chapter.

1.1 OVERVIEW

Medical imaging is the technique used to acquire visual representations of the interior of a body. It usage brings several benefits, including more effective medical services and treatments because it leads healthcare professionals to practice more evidence-based decisions[1]. It is possible to verify that a good management and a well performed medical imaging delivery and related services, provide an enhanced patient care[2].

The production of medical images had been growing notoriously in healthcare facilities during the past few years which creates new challenges regarding its storage, management and handling[3].

In medical imaging field, the data format, storage and transmission follow the standard Digital Image Communications in Medicine (DICOM). These files, besides the image or video, contain meta-data with reports, and other relevant data related to studies. The digital images, patient data, and studies are stored in local repositories addressing the concept of Picture Archive and Communication System (PACS). PACS encompasses the necessary hardware, software and communication infrastructure for acquisition, distribution, storage and analysis of digital medical images[4].

Several studies were performed in order to improve the PACS' communication services and mitigate the burden of maintaining such complex information systems. The time it takes to performed a certain search is a factor that severely deteriorates the overall PACS performance, especially when considering big data scenarios. Healthcare institutions do not have the resources to explore which is the best technology that meets their requirements and if the wrong one is chosen, it can bring several long term problems.

DICOM files are data structures very diversified that change according to the applied modality, practical and equipment, following standard templates that have mandatory, optional and conditional attributes. Depending on the desired use case, a relational approach can be hard to apply.

Over the last years, an open source PACS system has been developed by the BioInformatics, a research group from Instituto de Engenharia Electrónica e Telemática de Aveiro (IEETA) under the project "Dicoogle P2P Network". The Dicoogle architecture is a set of indexing and retrieval DICOM services that may be easily installed in a PACS server or a workstation capable of storing medical images in the DICOM format. The engine behind those services is designated Lucene, a plugin that uses the library Apache Lucene for its operations.

One big problem with the current Apache Lucene based solution is the performance (response time), mainly when the number of files starts to scale, which can compromise the desired workflow on data analytics and other operations where it is necessary to deal with a big amount of data.

1.2 GOALS

This thesis goal consists in the study, evaluation and development of distinct database solutions that can solve the existent performance issues on medical imaging in big data scenarios. Several technologies were analyzed, including relational databases, non-relational databases and text-based search engines, assessing the pros and cons of each.

The distinct solutions will be instantiated as index and query plugins on Dicoogle's platform, trying to be reliable alternatives to the current implementation. Performance measures will be taken, comparing results over different datasets with different sizes and number of DICOM files, following a strict test protocol to ensure that all the query plugins retrieve the same results, guaranteeing this way it correctness.

At the end of this thesis, we expect to have a new unified, scalable and reliable architecture prepared for big data environments on medical imaging, that can help physician's and researchers on their patient analysis/studies, keeping an acceptable performance for maintaining their desired workflow. Also, it will contain useful information for future PACS developers dealing with a similar problem.

1.3 OUTLINES

- ***Chapter 2:*** This chapter provides a description of the current state of art in medical imaging, including the DICOM standard, and an overview of PACS and it main services and models.
- ***Chapter 3:*** This chapter illustrates the technologies that can be used to perform the index/query tasks, including the relational databases, non-relational and free text search engines like Apache Solr and ElasticSearch, describing the advantages and disadvantages that they may bring in the proposed environment.
- ***Chapter 4:*** This chapter provides an overview of the Dicoogle's architecture and the previous work, including a brief description of how Apache Lucene based solution currently performs it index and query operations. It is also described the architecture that must be followed by the developed solutions (plugins), including the functional and non-functional requirements, where they are integrated, the main components and a detailed description of the used data models.

- ***Chapter 5:*** This chapter illustrates how the four explored solutions were designed and implemented, including the workflow of the index and query operations, how the query translations were performed and the applied optimizations over each engine.
- ***Chapter 6:*** This chapter presents the test results and how they were performed in order to obtain performance results on the index/query operations, which allow the validation of the developed plugins over the two identified data strategies. It is also presented the final architecture that we think that brings more benefits to the explained environment and that is the most suitable for replacement of the current solution based on Apache Lucene.
- ***Chapter 7:*** This chapter contains relevant conclusions and future work that can be realized in order to improve and explore new scenarios that arose as a result of the work of this thesis. The main contributions will also be presented.

CHAPTER 2

Digital Medical Imaging Laboratory

This chapter provides an insight into the current state of the art of medical imaging digital laboratories, including a brief overview of the medical imaging technique describing it systems and related technologies, such as PACS and DICOM. Also, it will be made an overview relatively to the Big Data scenario in healthcare institutions.

2.1 OVERVIEW

Medical imaging is the technique used to assemble visual representations of internal body structures hidden by the skin and bones, as well as to diagnose and treat diseases[5]. The arising benefits from it introduction are numerous, for instance, it provides a more effective medical care and treatment because it encourages healthcare professionals to practice more evidence-based decision making in clinical analysis and medical interventions. It is widely used for diagnosis, therapeutic and monitoring purposes. High-quality medical imaging is important for medical decision-making and it leads to the reduction of unnecessary procedures. For example, some surgical interventions can be avoided resulting of the analyze of the medical images.

Several subareas, such as Nuclear Medicine or Radiology are commonly associated with medical imaging because they generate the used medical images. The most well-known modalities are Computed Tomography (CT), X-Rays (CR), Magnetic Resonance (MR) and Ultrasounds (US). Medical imaging also establishes a database of normal physiology and anatomy to make it possible to identify abnormalities, by finding differences between the result and the standard[6].

Thanks to the rise of the digital era, the benefits when compared to the past analogical equipment are huge. However, the clinical staff has several adoption difficulties, refusing sometimes to learn how certain digital equipment works, declaring that they lead to control losses.

With the increased availability of medical equipment and improved healthcare policy, the number of global imaging-based procedures is increasing considerably. Nowadays, even small sized healthcare institutions use medical imaging systems. Consequently, it is necessary to add visualization and digital storage devices to improve the physician's workflow.

Moreover, healthcare institutions have purchased more acquisition devices, with higher resolution and larger image data size which requires that image repositories need to scale up in order to support such growing of medical studies.

2.2 BIG DATA SCENARIO IN HEALTHCARE INSTITUTIONS

Big Data can be defined as a collection of large and complex datasets which are difficult to process and can not be easily managed with traditional or common data management tools and methods [7].

The three Vs of big data can be expressed as volume, velocity and variety, composing a comprehensive characterization, which tells us that big data is not just about the size[8].

Regarding volume and as explained before, the number of imaging-based procedures is increasing considerably. The main reason is that standard medical practice is moving from relatively subjective decision making to evidence-based. Nowadays the professionals/hospitals are also more encouraged to use electronic health record technology. The development of new technologies such as new sensors, capturing devices, 3D Imaging and mobile applications and the fact that medical knowledge/discoveries are being accumulated also fuel this exponential growth.

The enormous variety of data-structured, unstructured and semi-structured is present in medical imaging. The data can vary, for instance, from modality to modality, being the generated DICOM files pretty different regarding organization and representing the big data concept.

Data is accumulated in real-time and at a high rate, creating new challenges with the unprecedented accumulating rate due to the big amount of sources and healthcare facilities applying medical imaging technique, addressing the concept of velocity.

2.3 PICTURE ARCHIVE AND COMMUNICATION SYSTEM

The volume of data generated, in medical imaging laboratories is huge, besides, redundancy is also crucial for data security. Efficient storage and permanent availability of all produced data are huge tasks. Institutions must deal with several Information Technology (IT) related problems, including fault tolerance, system scalability, performance issues, hardware maintenance costs, system obsolescence and migration. This huge amount of data usually requires upgrading hardware infrastructure and increasing costs over time to keep everything running with the right quality of service.

A PACS is a digital image system which provides storage and convenient access to images from multiple modalities (source machine types) and allows quickly access patients medical records. It tends to focus on providing a consolidated platform, so information becomes more readily available to physicians, whenever they need, and independently of their department location[9].

PACS encompasses hardware, software and communication networks for the acquisition, distribution, storage and analyze of digital medical images in distributed system environments. Electronic images and reports are transmitted digitally, which eliminates the need to manually file, retrieve, or transport by film jackets. The universal format for PACS image storage and transfer is DICOM that is described in the next section[10].

Figure 2.1 represents the three main sequential steps in medical imaging workflow.

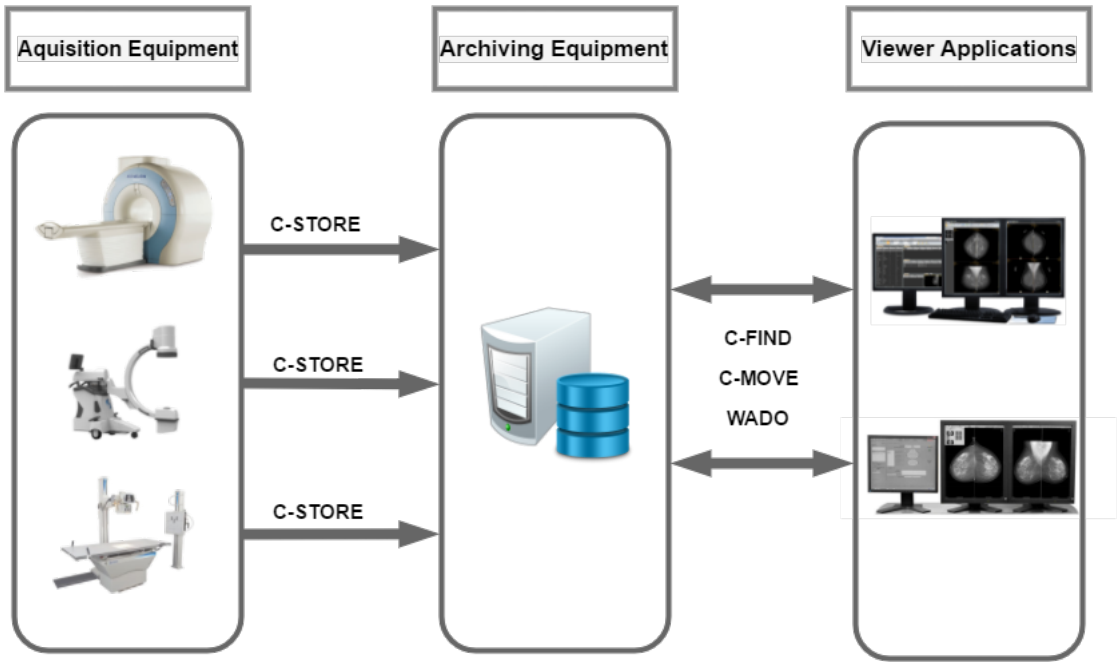

Figure 2.1: Major PACS Components. Based on[11]

These major components are described as image acquisition from devices (modalities), store image on a digital archive and image access by users at the viewing workstations[12]. All these components communicate through the network using the DICOM protocol, contributing to the integrated system. For a better perception of this components, a detailed description will be made next.

- **Acquisition** Can be defined as the process of image production (capture and digital codification) by equipment denominated as modalities. Medical images can be produced with two different ways. The first one is the film scanned by a laser, which allows compatibility with analogic equipment, ensuring that past studies can still be used. The other technique is when the digital medical image is produced directly from digital modalities equipment [13].
- **Distribution** Distribution refers to the process of moving images and associated data from a PACS node to another outside the sector. Distribution can be used to transfer images and associated elements among the components described above. Moreover, the PACS distribution process can reduce costs because it avoids the loss of studies, and it becomes easier to exchange studies with other institutions, allowing distributed workflows for clinical research, academic purposes, among others.
- **Visualization** Visualization can be described as the process through which users at the visualization systems (workstations) can actually use the PACS system. These workstations allow search by examinations, visualize images and meta-data, manipulate and send medical images to archive units. These workstations are typically used by the physicians to realize diagnosis. All available features must have a high quality of service, in order to avoid serious problems resulting from the occurrence of flaws.

The huge amount of data, created by the medical images (pixel data) and meta-data, leaves the PACS Archives Layer to deal with the storage. The PACS Distribution Layer also needs to keep the communication delays acceptable to the medical process. Besides that, medical imaging field imposes

several constraints to PACS implementations, including security procedures. Searching content in these repositories is not always trivial, and the proposed PACS architectures, must find the best trade-off between performance and features provided.

PACS workflow is usually divided into several stages, from patient registration in the Hospital Information System (HIS) and Radiology Information System (RIS), the examination procedure, image viewing by physician, reporting, and image archiving.

H.K. Huang[13] proposed 3 main architectures that can represent different kinds of workflows in a PACS: stand-alone, client-server and web-based model.

- **Stand-alone:** In this model, after the image acquisition, medical images are immediately sent to the image archive and then forwarded to the previously registered workstations. Workstations are able to query and retrieve studies from the archive server. One problem of this model is that images can be sent to more than one workstation, what may compromise the coordination among physicians, besides, can arise some security problems as studies are transmitted and can be accessed not only by strictly necessary personnel but virtually anyone within the PACS. The solution has several benefits, for example, there is less risk to loose studies because the modality can send images directly to workstations.
- **Client-server:** In this approach, all the repositories are centralized, this means that studies are uploaded from acquisition equipment to a central repository. Images are forward to PACS archive and, after that, the user chooses the required patient and download the correspondent exams. Because studies need to be downloaded all the time, the local network has to be very fast in order to this model work correctly. However, it provides a more efficient access control to studies than the previous architecture.
- **Web-based:** With web evolution, PACS got a new opportunity to evolve, making web-based architectures the current trend. In this model, the central archive is integrated into a data center, being available trough a Web interface. One potential problem is the fact that a network connection is needed. However, this brings a robust solution with security features, great portability. This model has been growing in the last years, leading in the present clinical workflow.

Nowadays digital medical imaging data can be generated in almost every healthcare facility, even one with limited human or financial resources[14]. Medical imaging equipment, e.g. X-ray generator, became gradually more powerful and less expensive, making them available in small imaging centers. However, a traditional PACS solution can require many resources with its IT infrastructure. This difference in resources, leaded hospitals and imaging centers, adhere to the outsourcing of PACS IT infrastructure to a remote data center, possible due to the evolution of the Internet and Cloud Based Computing. Because PACS repositories are vendor-neutral,PACS upgrading is not an issue. With this web-based PACS, inter-institutional data share is also easier because the solution can serve multiple sites.

Among the PACS use case scenarios, it is possible to identify two different purposes for it usage:

Traditional/Institutional PACS: It is the most applied by medical staff and used at healthcare facilities. Usually, has a less operational time window and it should have more performance at accessing content. Fewer requirements for content discovery features. The institutional PACS is protected by a high-security firewall with security requirements to ensure patient confidentiality.

Research PACS: Repository-wide searches. It is virtually impossible to reduce time windows. It requires content wide searches with the interest in all the data features necessary to satisfy the requirements of clinical research and clinical trials. The traditional clinical PACS is insufficiently flexible for this scenario[15].

2.4 DIGITAL IMAGE COMMUNICATIONS IN MEDICINE

DICOM is the main standard for storing, printing, handling, and transmitting information in medical imaging. National Electrical Manufacturers Association (NEMA) and American College of Radiology (ACR) formed a consortium with the intent of creating this standard for the technologies involved in the medical imaging field. Over time, this standard has suffered changes in order obtain a better response to requirements. The referred standardization brings a solution to several problems, for instance, it allows the exchange of studies among physicians using different vendors equipment because they need to follow the DICOM rules, resulting in a greater compatibility. DICOM is also a big step on the PACS improvement[16][17].

2.4.1 DICOM DATA FORMAT

DICOM files contain the image(pixel data), and the related meta-data header. These files can be divided into multiple data elements that form a DICOM Object. The header carries a certain number of fields according to the study modality. DICOM Information Model, defines a set of required fields, mandatory in every file in this format[18].

Figure 2.2 illustrates the main structure of a DICOM file.

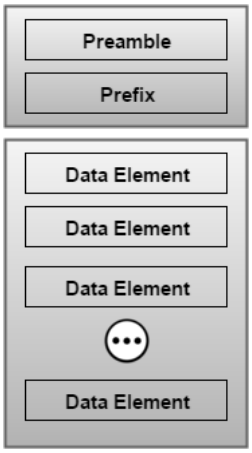

Figure 2.2: DICOM File Format. Based on[18]

The data elements that compose a DICOM file, are encoded using a Tag Length Value (TLV) structure, figure 2.3 explains what TLV structure means on a DICOM context:

- **Tag** This field, is a 16-bits unsigned integer, that identifies unequivocally the DICOM data element (group and element number).

- **Lenght** Gives information about the length (in bytes) of the value field.
- **Value** The Value field, is the binary data of the element, for example, the encoded string containing the patient name, or the pixel data from an image.
- **Value Representation** Value Representation(VR) is an optional field that has information about the type of element, for example, OB means Object Binary. This field is not mandatory because with a DICOM data dictionary provides also this information (for each tag the dictionary defines what type it stands for).

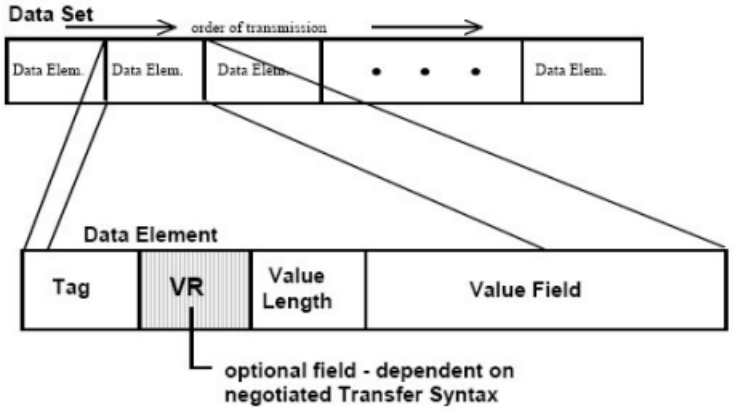

Figure 2.3: DICOM Data Element Structure. Based on[18]

DICOM objects are open to new tags definition, which increases its flexibility in terms of communication/functionality, for instance, private vendors can add their own data elements for their equipment. Also, some definitions are not required, depending on the applied scenario.

2.4.2 IDENTIFICATION AND HIERARCHY OF DICOM OBJECTS

DICOM objects organization follows a well-defined hierarchy that matches the real world. Like in real life several studies can be done on the same patient, these studies may be from different modalities, and each modality produce several images, in a DICOM object that hierarchy is also present.

Figure 2.4 illustrates the identified levels.

Each level object has Unique Identifiers (UIDs). Patient UID identifies the patient level, Study Instance UID the study level, Series Instance UID is the unique id at the series level. Each DICOM object is instantiated with a Service Object Pair (SOP) Instance UID, from the image level. UIDs are assigned to all DICOM objects, and is composed by <org_root>.<suffix>, where the <org_root> is the organization identifier, and the suffix identifies the actual object.

A SOP Class is defined as the union of DICOM Message Service Element (DIMSE), like Store, Get, Find, Move and a Information Object Definition (IOD). The SOP class definition contains the semantics and rules which may restrict the use of the services in the DIMSE service group or the attributes of the IOD[19]. A IOD is an object-oriented abstract data model used to specify information about Real-World Objects. A IOD provides communicating application entities with a common view of the information to be exchanged. A IOD also represents a class of Real-World Objects that share

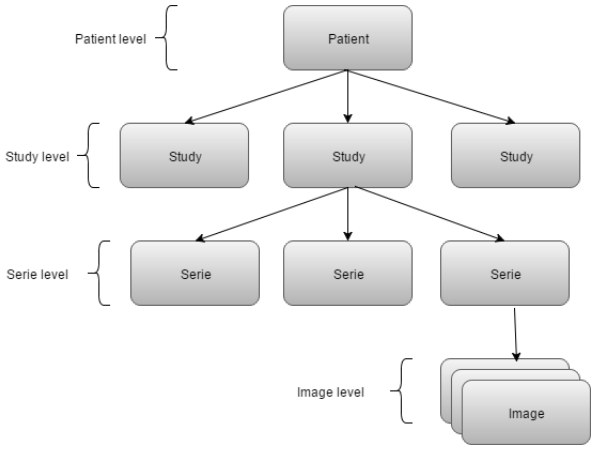

Figure 2.4: DICOM Information Model

the same properties[20], for instance, the MR IOD specifies the modules and elements of an image created by a magnetic resonance imaging device as shown in Table 2.1.

IE	**Modules**	**Elements**	**Usage**
Patient	Patient	Patient's Name Patient ID Patient's Birth Date Patient's Sex ...	M
	Clinic Trial Subject	Clinical Trial Sponsor Name Clinical Trial Protocol ID Clinical Trial Protocol Name ...	U
Study	General Study	Study Instance UID Study Date ...	M
	Patient Study	...	
	Clinical Trial Study	...	
Series	General Series	...	
	Clinical Trial Series	...	
Image	General Image	...	
	Image Plane	...	
	Image Pixel ...	...	

Table 2.1: MR Image IOD Modules

2.4.3 DICOM SERVICES

DICOM forms a set of different services, most of which involve the transmission of data over a network. These services are important for interoperability between medical devices.

Each service requires communication between a Service Class Provider (SCP) and a Service Client User (SCU).

A SCP is the DICOM node providing the DICOM service (server alike). A typical example of DICOM SCP is the PACS archive, that for instance, provides storage services to DICOM clients.

A SCU is the DICOM node using the DICOM service (client alike). This service class connects to the SCP in order to use/consume the DICOM service. Modalities and workstations are examples of SCUs, which usually use the storage services provided by a PACS server in order to send or retrieve DICOM images[21][22].

DICOM protocol works over TCP/IP and its messages are encapsulated using the TLV encoding. Each DICOM device has an Application Entity Title (AETitle), together with the IP and port number, allows its identification and communication within the network[22].

The first step in communication, called DICOM association, is a procedure where the devices negotiate parameters, including encoding, kind of information, image compression, and so on. After these steps, the service goal can be performed[23].

Services have several commands associated. The most exchanged ones are showed in Table 2.2.

Service	**Description**	**DICOM Command**
Verification	Check the status of DICOM devices	C-ECHO
Storage	Push image to PACS Archive	C-STORE
Query/Retrieve	Search and get the images	C-FIND C-MOVE/C-GET
Worklist management	Get work lists	C-FIND

Table 2.2: DICOM Services Description and Associated Commands

Next in this document, only the most relevant services for this thesis are described because they are the primary scope off the PACS image distribution layer and the most relevant in terms of typical performance issues.

- **Storage Service**
  The DICOM Store service is used to send digital images or other related digital data from a DICOM node to another DICOM node. In the figure 2.5 we can see the requests exchange flow and data that goes on each message.

  In a similar way with a client-server mode, C-STORE Request is invoked by the SCU, and all the DICOM objects are sent inside. After the SCP receives the message the file is received and the C-STORE the response is sent. The received response gives feedback about communicating success or failure of the storage request.

- **Query/Retrieve Service**
  The DICOM Query/Retrieve service is used, often by physician's viewer applications, to search and download (retrieve) from a DICOM archive to perform revisions on another DICOM node. This service can be divided into two phases, the query phase, and the retrieve phase. On the query phase a DICOM node (SCU) sends a request (which may include search parameters) to

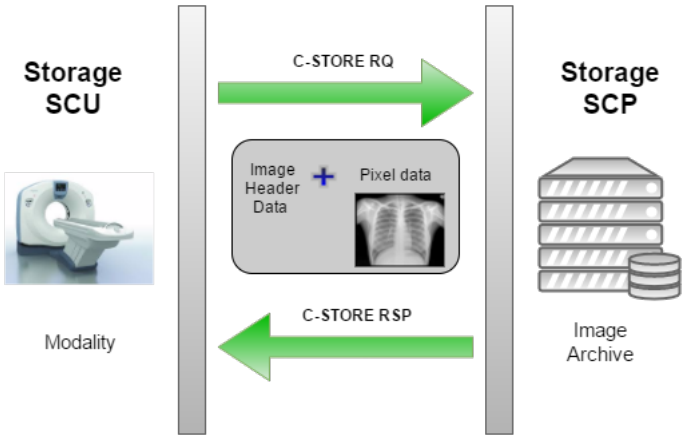

Figure 2.5: DICOM Storage Service

another node (acting as SCP) and expects the response. One example can be a workstation that queries a PACS server about the patient called Andre. The retrieve phase allows the SCU to get/move images from the SCP. After a result query, SCU chooses what needs to be transferred.

From the protocol point of view, the DICOM Query/Retrieve service is implemented through the C-FIND and C-MOVE messages. Figure 2.6 illustrates the query phase exchanged messages.

On the Query phase, the SCU sends a C-FIND-RQ (request) message to the SCP, including search parameters, and the SCP is expected to answer returning one or more C-FIND-RSP (response) messages to the SCU.

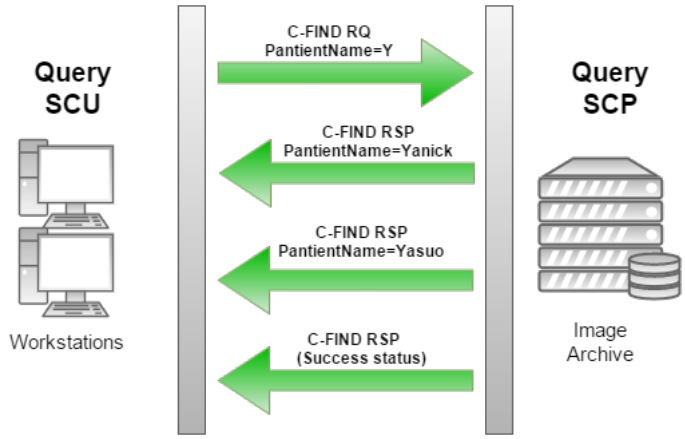

Figure 2.6: DICOM Query Service

During the retrieve phase, the SCU sends a C-MOVE-RQ (request) message to the SCP specifying the items to be retrieved. This request will trigger the transfer of the appropriate DICOM data through the DICOM Storage service (C-STORE messages). During the transfer one or more C-MOVE-RSP (response) messages are returned by the Query/Retrieve SCP to the SCU, providing updates about the status of the operation. Figure 2.7 describes the retrieve process.

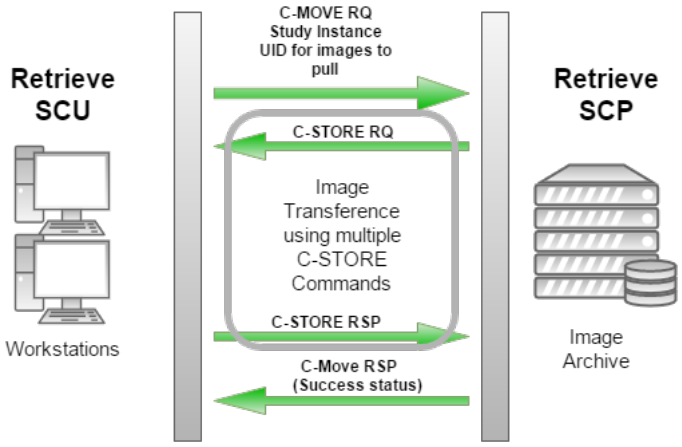

Figure 2.7: DICOM Retrieve Service

2.4.4 WEB ACCESS TO DICOM PERSISTENT

Web Access to DICOM persistent (WADO) is a recent web extension to the DICOM protocol. This standard specifies a web-based service for accessing and presenting DICOM persistent objects (e.g. images, medical imaging reports). This extension allows that objects are encapsulated in a HTTP/HTTPS connection and open new possibilities for integration with web applications.

One drawback of this service is that search and storage features were not included because is only possible to download a DICOM Object if it identifier is known.

It is intended for distribution of results and images to healthcare professionals. Objects are presented via HyperText Markup Language (HTML) pages or Extensible Markup Language (XML) documents, through HTTP/HTTPs protocol using UIDs[24], such as PatientID.

Figure 2.8 show the interaction between the client and the server.

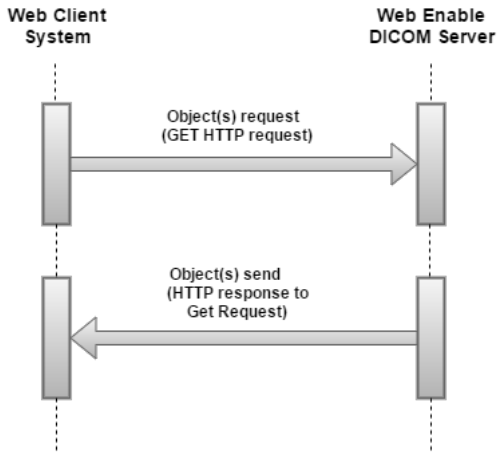

Figure 2.8: WADO Interaction Diagram

The HTTP GET method. follows the syntax : "http://<authority><path>?<query>" , e.g: "http://www.hosp.pt/dicom/wado.asp?studyUID=5". Some interesting applications of this service are:

- Referencing an image or a report from a patient record, for instance, send an e-mail in order to ask for physician's second opinion, with a referenced support image.

- Providing access to anonymous DICOM reports, images, and waveforms via a web server, for clinical trials or teaching purposes.

The last versions of DICOM standard (2016) has established data communication requirements with RESTful nature in order to replace the traditional communication directly over the TCP/IP protocol[25]. The main descriptions consists primarily of:

- **WADO-RS:** Restful version of the previously explained WADO, retrieving DICOM files and meta data in XML or JavaScript Object Notation (JSON) format.
- **STOW-RS:** Stores a specific instance into the server. The unique identifiers need to be provided.
- **QIDO-RS:** Allows query based on ID for DICOM Objects[26]. An implementation should support the search for studies, series or even instances. With a search parameter, it is returned a list of matching studies and the requested attributes.

CHAPTER 3

TECHNOLOGIES

This section provides a brief description of technologies that can be used to perform indexing and query tasks on PACS repositories. The main goals are study their performance in Big Data scenarios, implementing the identified use cases and find the most optimized solution. A database can be defined as a collection of information that is organized in a way that can easily be retrieved, managed, and updated. Following the technology progress regarding computer storage, memory and computer networks, the capabilities, sizes, and performance of databases and their respective Database Management Systems (DBMSs) have been growing in orders of magnitude. The development of database technology can be divided into two main categories based on data model or structure: relational and non-relational which will be deeply explained and assessed their pros and cons. Other trending technologies with great applications and performance in big data scenarios are the information retrieval systems, called free text search engines. Those technologies will be presented in the following sections and the trending/most popular will also be described.

3.1 RELATIONAL DATABASES

Relational Model has been proposed by E. F. CODD while working in IBM Research Laboratory, San Jose, California[27]. A relational database can be defined as a collection of data items organized as a set of formally-described tables, with rows and columns[28]. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows), for instance if a table has the columns username and password, all the rows will have usernames and passwords belonging to the same collection, with the same data types.

Data in tables can be related using keys, and this keys (primary and foreign) allows the retrieving of related data which is the basis for the term: *relational database*. The various software systems that work with relational databases are known as Relational Database Management System (RDBMS), and handles the way data is stored, maintained and retrieved. Virtually all relational database systems use Structured Query Language (SQL) as the language for querying and maintaining the database. They usually implement the four basic functions of persistent storage labeled CRUD operations: create, read, update and delete, for instance, updates on rows, delete tables, create indexes, etc.[29][30].

Relational Databases bring some relevant advantages when compared to other data store types. The most relevant are:

- **Reduced Data Redundancy:** With this model if some record needs to be changed, it is only required to change it in one place. This brings more efficient storage, with low data replication.
- **Complex queries granted with SQL language:** allow programmers a enormous amount of operations, including Insert, Delete, Create, Drop, etc.
- **Security:** This kind of databases, with data split into tables, makes possible to protect data through access limitations with keys. Because relational databases are older, security mechanisms respecting users are further developed and tested.
- **Data Model expansion:** It is easy to expand data model by simply add more tables and creating the needed relations[29].

This data model, where the structure of the data is predefined by the table layout and the fixed types of columns, brings some limitations, for example:

- **Scaling:** Users can scale a relational database by running it on a more powerful machine, with a lot of storage space. To scale beyond a certain point, it must be distributed across multiple servers and joining their tables across a distributed system can be hard, besides, this kind of databases are not designed to function with data partitioning, so distributing their functionality requires a lot of work and knowledge.
- **Complexity:** All the data that has to be stored needs to be converted into tables. When it does not fit into a table structure, storage can be a complex task, difficult, and slow to work with.
- **SQL on unstructured data:** If the data is structured, there are no problems with the data interaction, otherwise on unstructured data needs complex adaptation techniques[31].

The trending open source, cross-platform relation databases in today's market are MySQL, SQLite, and PostgreSQL[32]. A brief overview of each one of them will be presented.

3.1.1 MYSQL

MySQL developed by Oracle, current release 5.7, is the world's most popular open source RDBMS. MySQL is used by enterprises like Google and Facebook. This relational database follows a client/server model. Massive information regarding the database is available on the internet due to the popularity of the product and a massive active community. This available information helps developers to build more complex systems, with fewer effort.[33]

Table 3.1 shows the available data types that can be used to store values of DICOM elements. With few simple SQL statements is possible to create and interact with this database system. MySQL includes data security layers that protect sensitive data and features to improve response, for example, through the use of indexes. Java Database Connectivity (JDBC) is currently widely used and tested. Handles almost any amount of data, up to more than fifty million rows[34].

Data type	Description
INTEGER	Standard integer
BOOLEAN	True/False
DATE	Year, Month, Day
FLOAT	Double precision floating-point number
VARCHAR(n)	Character string. Fixed-length n

Table 3.1: MySql Data Types

3.1.2 SQLITE

SQLite is a software provided as a library and was developed by Dwayne Richard Hipp, that implements a self-contained, server-less, transactional SQL database. In result of being a database self-contained and file-based, SQLite offers a set of tools to handle data with fewer constraints when compared to server databases.

The communication with the database is made through direct calls to the file holding the data, instead of using the traditional sockets and ports from the client-server model.

Table 3.2 illustrates the available data types that can be used to store DICOM values. SQLite requires no installation before it is used, zero-configuration, just a single file that can be located anywhere.

This database allows users to store any value of any data type regardless of the column type. Some useful extensions to the SQL standard like "REPLACE" and "ATTACH/DETACH" are present.

The main problem of this system is the lack of features when compared with the remaining client-server based databases, including distribution mechanisms, fault tolerance, scalability, among others. Another potential issue is the fact that this database is stored as a single file, and a rogue process can easily open the database file and overwrite it, so security must be performed at the file level.

Data type	Description
INTEGER	Standard integer
TEXT	Text String
BLOB	Data stored as it was input
REAL	Double precision floating-point number

Table 3.2: SQLite Data Types

3.1.3 POSTGRESQL

PostgreSQL is an open source, currently on 9.5 release, object-relational database system developed by PostgreSQL Global Development Group. Table 3.3 show the data types that can be used to store DICOM values[35].

PostgreSQL, like MySql, also uses a client/server model. This system provides reliability and stability while keeping a good performance. Several programming interfaces are available, including Java, the most relevant for this thesis. The maximum database size is unlimited and it table size can grow up to 32 Terabytes, making this database, suitable for big data scenarios[36][37].

PostgreSQL is highly programmable, with customs procedures called *stored procedures* that are also available in other languages. These functions can be created to simplify the execution of repeated and complex database operations with better performance. Although this DBMS does not have the popularity of MySQL, there are many developed third-party tools and libraries designed to extend it.

Data type	Description
bigint	Signed eight-byte integer
boolean	True/False
date	Year, Month, Day
double precision	Double precision floating-point number
text	String with variable unlimited length

Table 3.3: PostgreSQL Data Types

3.1.4 APACHE HIVE

Apache Software Foundation developed an open source relational database labeled Apache Hive. It is a widely used data warehouse system for Apache Hadoop, which is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters, and it is currently used by many organizations in big data oriented applications due to its capacity for querying and managing large distributed datasets.

It provides an SQL-like language called HiveQL that is converted to MapReduce, used for processing and generating large data sets with a parallel, distributed algorithm on a cluster. Hive stores metadata in a configurable database like, for instance, MySQL or PostgreSQL[38].

Hive Thrift Client connects to the running Hive server in order to perform client calls that also are available through several database drivers. Hive looks like traditional database code with SQL access. Because Hive is based on Hadoop and MapReduce operations, there are several differences when comparing with the previous databases. The most important one is that Hadoop is intended for long sequential scans, meaning that if the application requires small query response times, this database is not recommended.

Because this is a very important factor, this database was not selected. However, it performance on distributed scenarios can be explored in eventual future work.

3.2 NON-RELATIONAL DATABASES

A non-relational database (Non Structured Query Language (NoSQL)), sometimes referred as document-oriented databases, provides a mechanism for storage and retrieval data, that is modeled in non-tabular means different from the ones used in relational databases.

This kind of databases have existed since the late 1960s but did not obtain the "NoSQL" moniker until the increase of popularity in the early twenty-first century, triggered by the needs of Web 2.0 companies such as Google, Facebook, and Amazon.

NoSQL databases use storage systems with different goals. These databases have weak data consistency models, providing data reliability and availability through high-level software features and protocols[39][40].

The new data management systems used, are designed to support different workloads compared to traditional relational database systems (RDBMS), optimized for frequent updates with transactional processing. The workloads of these new systems are dominated by high-throughput append-style inserts rather than frequent updates of multiple values in a single transaction and read accesses mostly to perform queries for a single value. As a result, the data access patterns generated by these new systems can be quite different from traditional DBMSs.

NoSQL databases are divided into four distinct categories[41]:

- **Document-oriented:** The databases currently using this paradigm, have their architecture based on Documents. These documents can be XML, JSON, Binary JSON (BSON), and so on, following a tree hierarchical data structure, for instance, collections[42]. Databases implementing this paradigm, are focused in big data storage and good query performance.
- **Column-oriented:** Each database table column is stored separately, with attribute values belonging to the same column compressed stored and packed, while traditional database systems store entire rows followed by the other [43]. Column groups are sets of related data that is often accessed together.
- **Key-Value:** In these databases, a Value corresponds to a Key. This simple structure increases the query speed when compared to the traditional relational database, supporting mass storage, and high concurrency queries[41].
- **Graph Databases:** In Graph Databases, the stored entities are instances of objects, also called nodes, grouped with their attributes and relationships between these entities. The relationships are called edges, and can also have attributes. The stored data can be interpreted in different ways, based on the relations [44].

Some advantages of the non-relational model while comparing with relational databases are obvious, including:

- **Flexible data models:** Minor changes to the data model of a RDBMS require careful consideration and may necessitate downtime or reduced service levels. On the other hand NoSQL databases have almost no data modes restrictions, meaning that it is possible to store virtually any data element structure. Application changes and database schema changes are not difficult to be managed[31].
- **Horizontally scale over many commodity servers:** In the database world, horizontal scaling is often based on the partitioning of the data, this means that each node (server) contains only part of the data. Almost every NoSQL database are prepared with features that allows to perform easily this data partition, meaning a transparent expansion to take advantage of new nodes[45].
- **Fewer admin tasks:** The design/implementation of a relational database, is a much harder task that on NoSQL case. NoSQL databases bring features like automatic repair, data distribution, and simpler data models lead to lower administration and tuning requirements.
- **Better performance on Big Data scenarios:** NoSQL databases are often faster because of simpler data models. With no technical requirements (constraints) like the ones that relational databases have, most NoSQL systems can have much better performance.

This data model also brings several disadvantages, for instance:

- **Maturity:** RDBMS systems have been around for a long time, NoSQL databases are newer. RDBMS systems are richly functional, stable, with a lot of compatible tools. Many important features in NoSQL databases are still in development. Also the support can be compromised[31].
- **Overhead and Complexity:** Manual query programming is required (no SQL), which can be fast for simple tasks but heavy for others. In addition, complex query programming can be harder[45].
- **No Atomicity, Consistency, Isolation, Durability (ACID) transactions:** NoSQL databases do not have native support for ACID transactions. This this can compromise data consistency unless manual support is provided. Not providing consistency enables better performance and scalability but is a problem for certain types of applications that require their usage. Also it is not possible to perform atomic operations in multiple collections but only over a single document. The isolated and durable concepts are also only partially applied. [46].

The trending NoSQL are MongDB, Cassandra, Redis Database and HBase described in the next sub-sections.

3.2.1 MONGODB

MongoDB is a multiplatform and open source NoSQL database, following the document-oriented paradigm, developed in C++. [47]

The documents are stored in collections according to their structure. However, they can have a different structure in the same collection with negative impact on the performance. The documents are denominated BSON documents, with a maximum size of 16MB each. BSON are extensions of the JSON to allow increased number of data types and to provide encoding and decoding among different languages.

Each document is identified by the field with the tag "_id" and over that field is created a unique index. Although indexing is important to execute efficiently read operations, inserts can be slower. Additional indexes can be created over several fields or sub-field of the documents within the specific collection.

MongoDB supports powerful query language, allowing users to perform almost the same queries that the ones over a table on relational databases[48].

Figure 3.1 illustrates a typical JSON document that can be stored in MongoDB.

MongoDB makes use of a readers-writer lock mechanisms to allow concurrent data access.

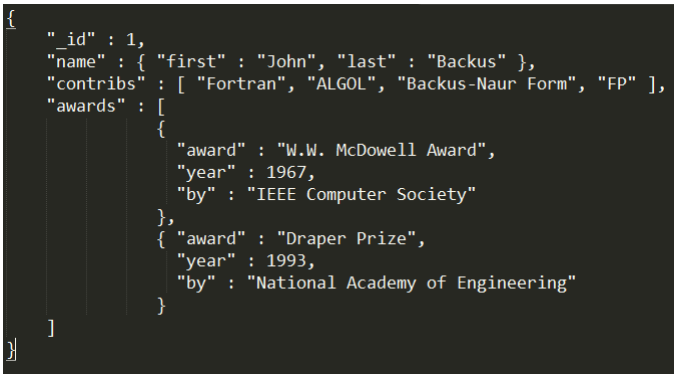

Figure 3.1: JSON Sample from MongoDB Official Website

3.2.2 CASSANDRA

Cassandra is also an open source NoSQL database developed by the Facebook group [49]. This database is referred as a "column-oriented" database. Data stored can be structured, semi-structured or even unstructured. Like MongoDB, Cassandra is designed to store a large amount of data in an efficient way. It is possible to store till petabytes of data[41][48]. Also, the schema is flexible, allowing operations like file removing and addition with simple operations.

Column family is an important concept and can be defined as a container for an ordered collection of rows, and each row is an ordered collection of columns. In Cassandra, although the column families are defined, the columns are not. Making possible adding columns at any time.

The main difference between a column family and a relational table is: Once defined the columns for a table, on each row all the columns must be filled at least with a null value, on a column family, any column can be added to any column family at any time.

Cassandra also is a distributed database with great fault tolerance. If a node affected, another one can give the required data. Cassandra also supports rich data structure and powerful query language. Figure 3.2 represents a JSON of how a column family definition look like.

```
UserProfile = {
    John = {email:"John@cassandra.com", phoneNumber:916424214},
    Terry = {email:"Terry@cassandra.com", fax:234552289},
    Matheus = {email:"Matheus@cassandra.com",gender:"M", address:"Lourical"},
}
```

Figure 3.2: JSON-Like Notation of Column Family

3.2.3 REDIS DATABASE

Redis is an open source (BSD licensed) written in C, currently in version 3.2.0. It is a non-relational database, following a key-value approach[50]. In addition to database usage, it also can be used as cache or message broker. Several data structures are supported, such as strings, hashes, sets, lists and so on.

Redis has built-in replication, implementing automatic partitioning using Redis Cluster, Lua scripting, etc. This database has an in-memory data structure store, meaning that when Redis starts all data is loaded into memory, so all operations are performed over the data structures existence on

memory[51]. This leads to an interesting performance and can be useful in several scenarios. However, it can not be applied to this thesis scenario because the amount of supported data is limited to the existing physical memory, which is not enough.

3.2.4 HBASE

HBase can be defined as a database which follows the non-relational paradigm with a database model designated wide column store, that is the same model used by Cassandra database[52]. This open source database is currently in version 1.1.4 and was developed using Java Language, by the Apache Software Foundation.

The storage is logically organized into tables, rows and columns. It is based on the Apache Hadoop, allowing horizontal scaling, and on BigTable concept which is a data storage system built on Google File System[53]. Their columns do not require a pre-defined schema and a row is a grouping of key/value mappings identified by the row-key.

HBase is still under development with only a few committers working on it, resulting in some crucial features still under development. It queries are written in a custom language that needs to be learned and has no native support secondary indexes[54].

3.3 FREE TEXT SEARCH ENGINES

A free text search engine is an information retrieval system, designed to find a required data stored in a computer system. This includes two distinct tasks, index and query/retrieve.

These engines apply text indexing techniques, in order to reduce the time required to find information and return it, in a format called search results, usually presented in a list, called hits[55].

Full-text queries make possible to search for a set of given terms in a set of documents. In fact, with text data type, this engines performs usually better.

Full-text search engines differ from databases mainly in terms of the organizational structure of the data (data model). One big difference between relational databases is that search engines are document oriented while relational databases are table oriented with lookup keys and a join capability for querying between them.

They are different from document-oriented databases because they handle distinct data models, for instance, text-based search engines (e.g. Solr), have flat documents structure, like a table, but with no multivalued fields (field with an array of values). NoSQL databases are different because they can have nested documents.

Index/Search are also operations with a lot of differences features, like the ones described in the next sections.

3.3.1 APACHE LUCENE

Apache Lucene is one of the most popular, open source, full-featured, text-based search engine[56]. Lucene provides application programming interfaces to perform common search and search related

tasks. Figure 3.3 illustrates it main architecture [57] [58].

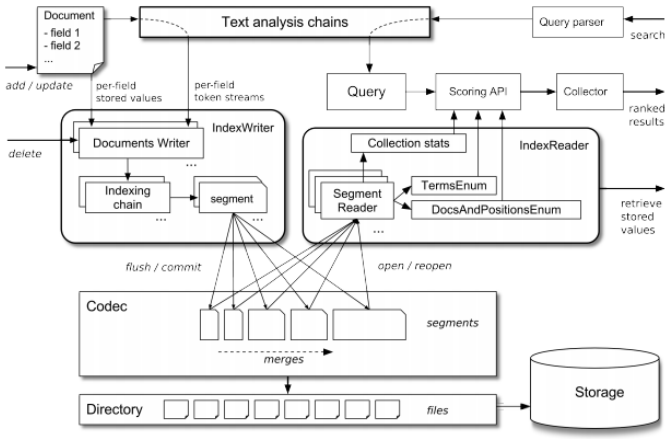

Figure 3.3: Apache Lucene Architecture. From [57]

The main features of this search engine are: Ranked searching, several query types, fielded searching, sorting by field, multiple-index searching with merged results, concurrent update and search, highlighting, suggesters and configurable storage engine.

Index task can be divided into three main steps: *Conversion to text*, *Analysis* and *Index writing*. To index data with Lucene, data needs to be converted into a stream of plain-text tokens (conversion to text), the data format that Lucene can accept. These tokens are used to populate Field instances. Field methods always take *String*, *Date* and *Reader* values. The data has to suffer pre-processing to avoid the index of things like HTML tags or XML elements. The next phase is *Analysis*, where the documents populated with fields are analyzed to make it more suitable for indexing, splitting the textual data into chunks, applying optional operations on them. For instance, the tokens can be lowercased to make searches case-insensitive and the stop words can be removed. The last task, *Index writing*, Lucene stores the input in a data structure called inverted index.

Inverted index makes efficient use of disk space while allowing fast keyword lookups. This index is a dictionary that records which documents contain a given token, when a document is given to the indexer of the search engine, the document is split into a list of tokens. These tokens are inserted into the dictionary and the document id is added to the right list. This indexing strategy, allows faster results retrieving for a given query, because when a user performs a full-text search query, the search engine finds the inserted terms in the dictionary and collects the lists with the document identifiers [59].

With a ranking algorithm, the best document matches are found. Lucene index also allows access to multiple threads and processes and has a locking mechanism that prevents concurrent index modification [60].

Figure 3.4 illustrates the structure of an inverted index.

Solr and ElasticSearch (described in sequel) are two open source projects based on Lucene.

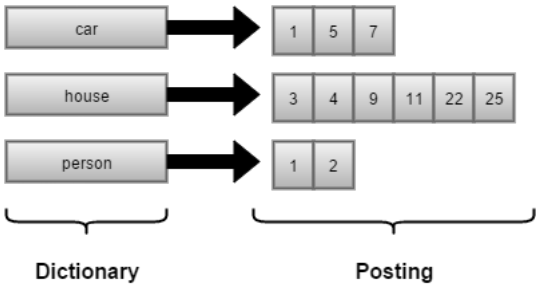

Figure 3.4: Sample Inverted Index Structure

3.3.2 APACHE SOLR

Apache Solr is as an enterprise Lucene-based search server, written in Java. Solr is used by sites like CNET, Zappos and Netflix. Hypertext Transfer Protocol (HTTP), XML and JSON are the standards through communications are established to the Solr server.

Beyond inherited Lucene features, Solr adds several features like:

- **Server HTTP:** Solr has a server that communicates over HTTP using formats like JSON and XML.
- **Configuration files:** Indexer schema, fields, text analysis configuration are defined on several XML files.
- **Caches:** Faster search are possible on result of several types of caches.
- **Facet search:** Features added on facet search.
- **Web interface:** Interface for administration tasks, like performance statistics, debug, index selector[61]. Figure 3.5 is a screenshot of the web interface.

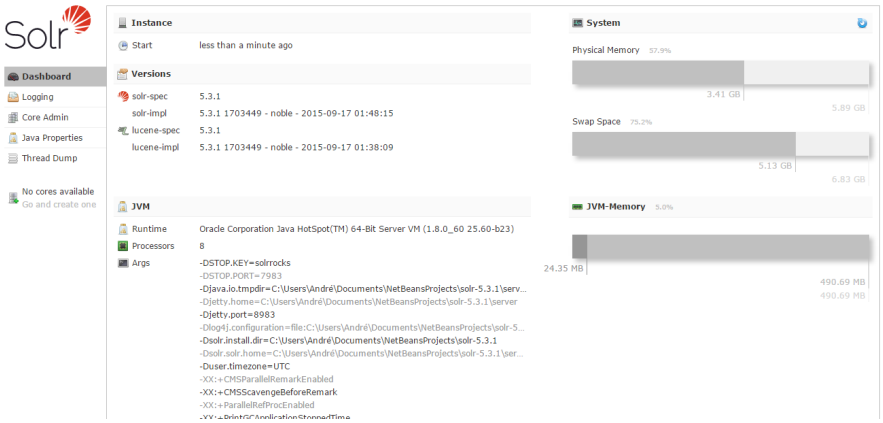

Figure 3.5: Solr Web Interface

Solr has a wide deployment and active developer community, with an official client library for Java. Release 4.4 comes with schema-less feature so that user do not have to build the schema with all the documents fields. With arguments from table 3.4, a lot of features can be achieved by the query method.

Parameter	Default value	Description
q	(configuration files)	The query
start	0	Offset into list of matches
rows	10	Number of returning results
fl	*	Fields to return
qt	standard	Query type to handler
df	(schema)	Default field to search

Table 3.4: Sample Query Arguments Apache Solr

3.3.3 ELASTICSEARCH

ElasticSearch is an open source search server developed in Java, also based on Lucene, started by Shay Banon. It provides a distributed full-text search engine with an HTTP web interface and schema-free JSON documents. Some statistics websites (http://db-engines.com/) show that ElasticSearch is one of the most popular search engines. The basic concepts related to ElasticSearch are:

- **Index** An index is a place where the data is stored.
- **Document** A document is a set of fields, that can occur several times (multivalued). A field can contain other sub-documents or arrays. It can have a well-defined structure with a schema or not.
- **Document type** Document types help to differentiate objects.
- **Node and cluster** ElasticSearch works in two different modes, standalone (single-search server) or with one or more cooperating server. One single Server is called node, and a group of nodes forms a cluster. With index splitting (sharding) large amounts of data can be split across several nodes.
- **Shard** It can be defined as a piece of data. This allows to overtake Random Access Memory (RAM) limitations, hard disks capacity and also by splitting data into several nodes.
- **Replica** It is a shard that increase query throughput or achieve high availability. When the primary shard is lost, a cluster can promote a replica to be the primary shard[62].

Elasticsearch is distributed, which means that indexes can be divided into shards that can have replicas. Each node hosts one or more shards and acts as a coordinator to delegate operations to the correct shard. With schema-free features, makes it easier to create and implement large search systems.

3.3.4 SPHINX

Sphinx, like Solr and ElasticSearch, it is a free open source engine that enables full-text search functionality to client applications. It can be used to communicate with other DBMSs by using native protocols of MySQL, MariaDB, and PostgreSQL, or by using Open Database Connectivity (ODBC) with compliant databases [63]. Official implementations of the Application Programming Interface (API) are available for PHP, Java, Perl, Ruby and Python languages.

Sphinx's index format generally supports up to 256 fields. The text sent to Sphinx gets processed, and a full-text index is built from that text. Additional values called attributes associated with each

document used to perform additional filtering and sorting during search operation are not added to the full text indexed but stored with the same properties.

Searches can be distributed across multiple machines, enabling horizontal scaling. Because Sphinx was written in C++ it does not run on all platforms while comparing with Solr and ElasticSearch which run on a Java virtual machine[64].

CHAPTER 4

Dicoogle System

This chapter seeks to explain how this dissertation work is related to the Dicoogle PACS and introduce the architecture of the developed solutions. First, some related studies will be described. These studies are important because they help in PACS development or they studied the same or similar problems. Furthermore, the Dicoogle platform will be described, including its architecture and a brief description of the index and query tasks and how it interfaces work. An overview of the existing Apache Lucene based plugins will be made. It also contains an insight into the system architecture of the developed solutions for the indexing and query operations in Dicoogle's platform, presenting the system functional and non-functional requirements that the developed plugins need to follow. The plugin's software architecture and the data model, including the applied database schema and the document structure chosen will also be presented.

4.1 Related work

Over the last years, several studies have targeted the performance of medical imaging repositories, their main goal was of keeping them fully operational even when the amount of generated data increases significantly, including new storage strategies, the way that communication is processed and other improvements on operations that can deteriorate the global performance.

Langer et. al. explained the application of cloud computing for solving some problems faced by medical imaging researchers, whose work include the data mining and processing of DICOM data[65]. However, the proposed scenario is different from ours since we target not only the research PACS but also the institutional, which are mainly focused on serving the medical practice. It also focus on improving storage features rather than the optimization of content discover.

Other studies have targeted the importance of optimizing the performance of the PACS operations, especially in distributed environments. In [66], the authors proposed a routing mechanism, and an efficient cache technology for distributed medical imaging scenarios that particularly targets the delays introduced by these scenarios in medical imaging studies retrieval. In [67] the same problem was discussed and an improved communication protocol is proposed.

The study referred in [68] suggested an automatic framework designed to optimize the workflows associated with distributed PACS scenarios. By leveraging multiple indicators associated with the

radiology workflows, it was possible to predict whether a given study is likely to be requested, and anticipate its transference procedure so that the study is already available when requested. Although a successful prediction can bring several advantages, the idea of an optimized search works as a complement that maximizes the overall system performance.

Alessandre et al. proposed a data storage method where the DICOM attribute values are stored in a schema on top of a standard relational database management system[69]. Although they were able to improve their PACS performance, they reached this conclusion using a minimal dataset of 3GB data volume containing only 35 studies, representing less than a normal work-day in small/medium sized institution and, of course, applied to a different PACS. So, they did not provide a scalability assessment, and it is well known that the performance of database technologies tends to decay over time as load increases. On the contrary, our approach uses datasets with millions of images from multiple modalities to produce a more accurate assessment of performance and scalability over time.

4.2 DICOOGLE PACS

This section will describe the Web PACS Dicoogle, including it architecture, main services, the existent Software Development Kit (SDK), their index and query functionalities and finally it will be briefly explained it current solution used to allow these functionalities based on Apache Lucene.

4.2.1 ARCHITECTURE AND SERVICES

Dicoogle is a software framework which allows developers and researchers to quickly prototype and deploy new functionality, taking advantage of the existent embedded DICOM services. It is an implementation of a PACS archive very amenable to extension due to its plugin-based architecture which helps developers to access DICOM datasets and metadata, detection of inconsistencies in institutions data and processes among other relevant features. Its plugin-based architecture enables different features to be developed separately and easily integrated.

Custom builds are done by including only the desired plugins[70]. Their lifecycle is managed by the Dicoogle core, being possible to enable and disable each module per user request leading to a minimized area of impact of experimental components. It allows Peer-to-Peer (P2P) communication models within a DICOM network. Dicoogle full stack architecture is illustrated in Figure 4.1.

A SDK was created in order to simplify the development by third parties and assure a full compatibility. With the implementation of the available interfaces, after generating the *.jar* file and insert it in the correct directory, it is loaded by the Dicoogle's core on application startup. Five main functionalities are present, including storage, indexing, querying, service and presentation.

The store service provides the persistence mechanisms and URLs to identify a resource (DICOM File), and it has the methods for retrieve a DICOM object matching a certain URL and also store any object or stream returning the correspondent URL. The service category, contains the most important DICOM services, including C-STORE, C-FIND and C-MOVE, and WADO. Given this thesis theme, only indexing and querying categories will be explained in detail although the others are used directly or indirectly.

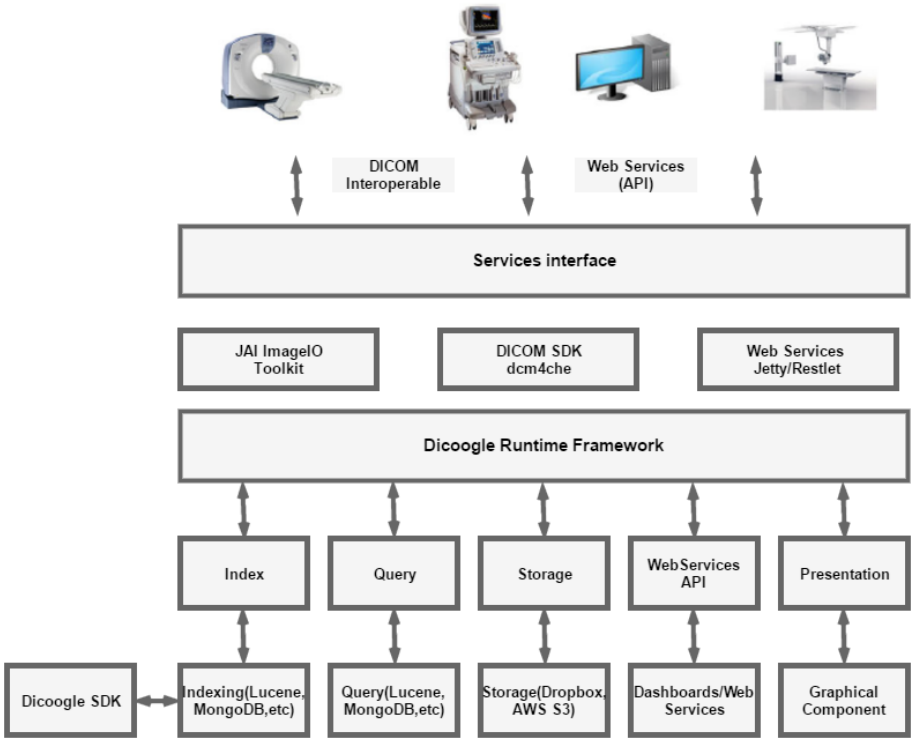

Figure 4.1: Dicoogle Architecture. Based on [70]

4.2.1.1 INDEX FUNCTIONALITY

Indexing plugins, organize data in a format that allows quick access, which includes the processes of data extracting and storing. Data models used are the ones defined by the plugin developer. More details of this operation are illustrated in Lucene Plugin section because the data flow and processes are quite similar. In order to call the index method, there are two distinct ways available, one of them through the web interface and other through the web services API. After the path to the folder containing the DICOM files is selected, the index tasks starts. Upon finish, a report is showed with some important aspects, for example, the number of indexed files or errors and the elapsed time.

4.2.1.2 QUERY FUNCTIONALITY

Query plugins, allow the access to information, through retrieve methods. Those methods convert the data structures, that fill the user requirements in a given query, to data representation understood by the Dicoogle front end. This means that, for each index plugin, needs to exist other that can make this conversion.

Dicoogle's web interface support free text queries, for instance, to find all the studies from 2016 from male patients the query is: "PatientSex: M AND StudyDate:[20160101 TO 20161231]". This query format requires when using other plugins different from free text search engines or the do not have Lucene query syntax support, a query transformation in order to adapt current Dicoogle standard.

4.2.1.3 INTERFACES

The first important feature displayed to the user in the Web interface is the call of the method for indexing a set of DICOM files. By inserting a path to the directory, the index method from the working indexing plugin is called, giving real-time feedback about the operation status, including the number of indexed files, the error occurred, and the elapsed time.

Another core feature is the possibility of querying an index. After a query submission, the current working query plugin is called. The results are returned, assuming that the data is correctly indexed, at the Instance level, which means that every single retrieved document contains information about which Series it belongs, which Study, and also the Patient relevant data, besides the image Uniform Resource Identifier (URI) and other selected set of fields.

This information needs to be converted and the results are displayed in a hierarchical way, allowing users to select first who is the patient that they are looking for, then select the study, the modality (Serie), and finally choose which instance in order to visualize the image, dump it, among other available operations.

Besides assuring compatibility with the Web interface, the developed plugins also need to work with the available Web Services, in order to enable features required for a research PACS and to be used by external applications. Further details will be elucidated throughout the Chapter 5.

To add an insight about the platform, Figure 4.2 illustrates a possible output from the query *PatientName:Felix* performed through the Dicoogle's interface.

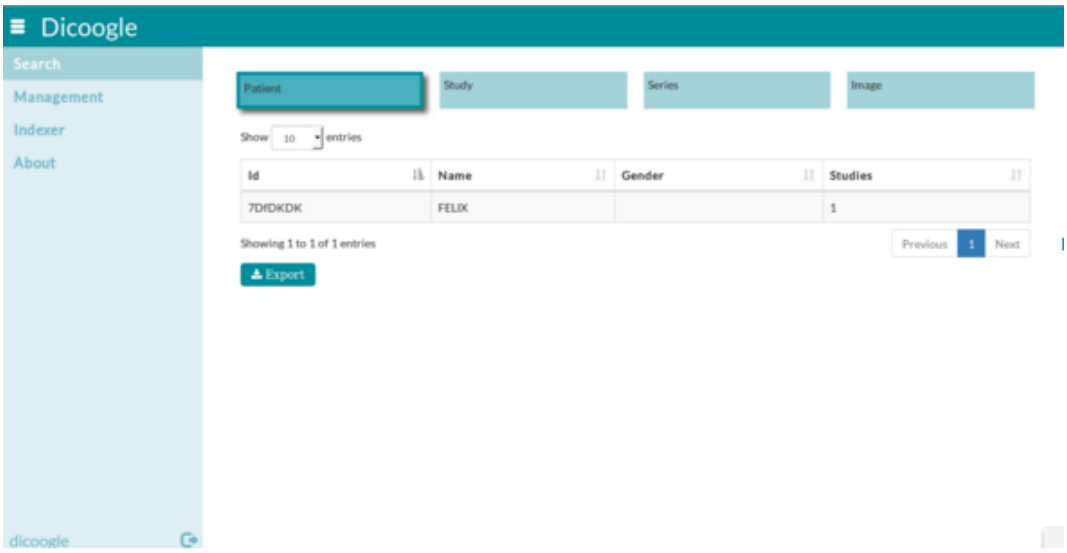

Figure 4.2: Dicoogle Interface Sample Query

As we can see on the application screenshot, the results are presented with the four levels from the DICOM information model previously introduced.

4.2.2 LUCENE PLUGIN

As referred before, a solution using Lucene is the current main plugin to perform index and query in order to process DICOM requests. For that reason, a short description of how it works will be made.

A typical DICOM persistent object usually contains several data elements, like pixel data, structured reports, and even free text. In addition contains descriptive attributes, such as image resolution, equipment reference, etc. A typical PACS only stores a minimal dataset with mandatory fields for each DICOM Information Model (DIM) level and so, many information sometimes important is lost, making it unavailable to be used in some data analytics tasks. Dicoogle Lucene Plugin follows a document-oriented paradigm, making possible to index any type of document and add other DICOM attributes without the need of creating new fields, tables, and relations[71][72].

Figure 4.3 illustrates is main components and methods.

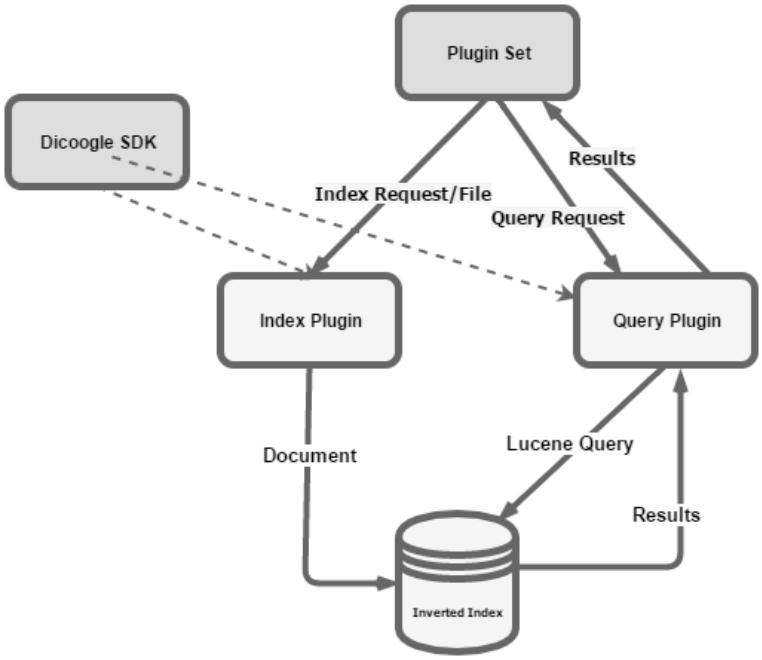

Figure 4.3: Lucene Plugin main Components and Methods

With the plugin-oriented architecture, some standards present in Dicoogle SDK need to be followed. The indexer and query components are implementations of the Indexer Interface and Query Interface from the SDK.

4.2.2.1 INDEX METHOD

On index method call, a structured containing all the file read is converted into a data structure that is a list of DICOM elements. After that, the data structure containing the fields is converted into the Document structure from the Lucene library using de TLV encoding. When the method that adds a document is called from the Lucene writer Object (after it correctly configuration), the document is indexed. Every file indexed with Lucene relies on transactions, which allows roll-backing if some error occurs.

4.2.2.2 QUERY METHOD

On the search side, Lucene supports a variety of query options, along with the ability to filter, page, sort results and so on. Given a certain query, the query method returns a *Iterable* containing the

different results. In order to perform such task, several objects from Lucene library are instantiated and filled with the necessary meta-data, including the fields that need to be retrieved.

4.3 PLUGINS

This section illustrates rules that must be followed by the developed solutions (plugins) and explains where the developed components will be integrated. The data models applied will also be described.

4.3.1 REQUIREMENTS

On the digital medical imaging world, certain requirements are different from the ones generally applied on this data storage and query scenarios must be followed. This section illustrates the most important ones, taken into account while developing the several solutions presented in chapter 5. The system's requirements can be divided into two different categories: functional and non-functional. The functional requirements define what the system should do, on the other side, the non-functional describe how the system should behave.

4.3.1.1 FUNCTIONAL REQUIREMENTS

In this thesis work, it is possible to identify several crucial functional requirements for keeping the developed plugins fully compatible with the current Dicoogle's standard.

The first one is the capability of being configured using a file containing user specified settings. Variables like data folder location, databases names, and other important application/environment settings, need to be present on a file using XML notation. These setting are applied after the plugin initialization.

Another important requirement is the use of transactions in order to keep the data integrity, assuring that the DICOM information model is secure, for instance, if an error occurs while inserting a given study, the images of that study can not be indexed/stored because all images require a study parent.

Besides all the methods necessary to connect to the database, and execute the necessary commands to query/index a given DICOM file, it is necessary to implement the methods from the Table 4.1, in order to ensure a fully functionality and needs to be present on all developed solutions.

Method	Description
enable	Check if the plugin is ready to perform an operation
disable	Check if the plugin is not ready to perform an operation
getName	Returns it name so it can be used as provider
setPlatformProxy	Allows the plugins to call operations from the dicoogle core
setSettings	Automatically called by the core to setup configurations
handles	Check if it has support for index a given file
unindex	Deletes the current index

Table 4.1: Extra Necessary Methods of Dicoogle Index/Query Plugins

Regarding operations' feedback, the main methods of index/query have to return a report containing relevant data like the elapsed time on a given query, the number of results returned, the number of indexed files, errors that occurred (e. g. usually the DICOM's directories contain files in a non DICOM format) and even the elapsed time of the index operation among others. Given the big data scenario, the RAM memory used on a task needs to be controlled, in order to prevent an error caused by memory overload.

4.3.1.2 NON-FUNCTIONAL REQUIREMENTS

First, the elapsed time to perform a certain task is probably one of the most important non-functional requirements. One the index task, despite that the system users tolerate some delay when the number of files grows exponentially, the spent time can be a critical issue and dissuade users of performing this operation. Regarding the query task, it is crucial for any search system, to return results on the faster possible way.

The retrieved documents for a given query must match the required information by the user, allowing the appliance of several filters and other extra features available on traditional search systems.

Another important requirement is the number of required configurations that should be low. Given the big diversity of users, they can lack the knowledge to properly configure the system.

As referred before, when dealing with DICOM files, privacy issues are very important and security measures have to be taken all the time, in order to keep the data protected as much as possible.

The last main non-functional requirement is the amount of disk space required. Although the storage devices price is lower that ever, on big data it can still be an issue if the size of databases/index is a lot bigger than what it should be and it can dissuade replication strategies and other important operations.

4.3.2 ARCHITECTURE

This section presents general aspects, significant elements, and other general proprieties used to design, modeling and subsequent implementation of the developed solutions.

Dicoogle's platform as referred before is based on a plugin-oriented architecture, this forces plugin developers to follow several strict rules in order to allow a full integration with the desired consistency leading to a system working properly.

For a better perception, the diagram presented on Figure 4.4 describes the main components presented in the Dicoogle's architecture where the developed plugins are integrated:

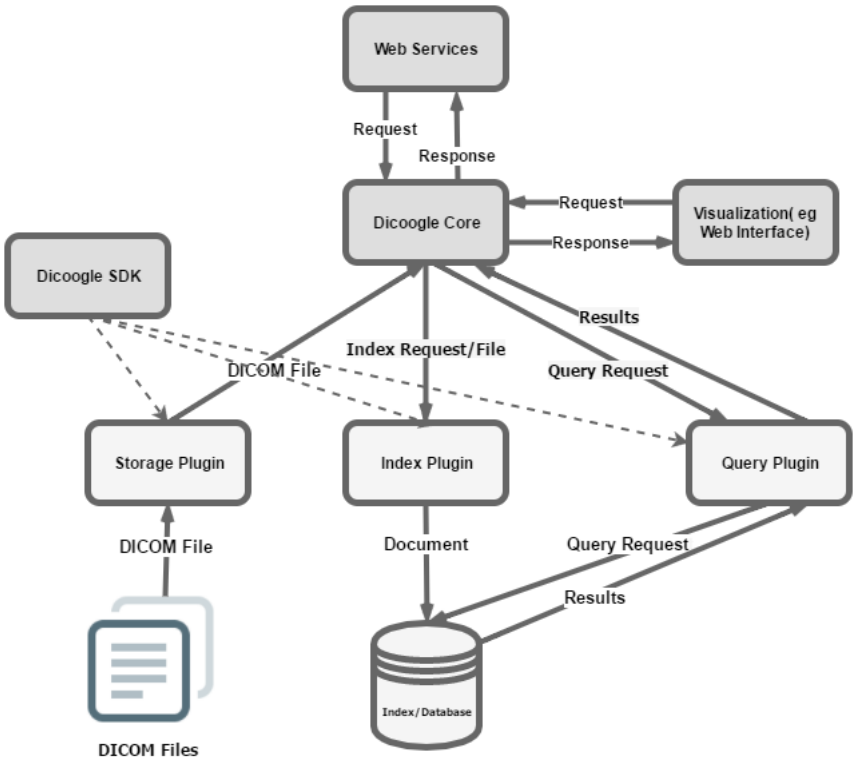

Figure 4.4: Global Software Architecture

The first component to start is the Dicoogle's core. This run-time framework allows the load of third party components and modifies or enhance its core functionality, including storage, index, and query. Communication, services, and lifetime are all managed by the core application. Besides the traditional PACS operations, web services API provides several features like search, dump to a DICOM file, export query results to a Comma-separated values (CSV) format, among other useful operations.

Dicoogle's core allows each module to be enabled or disabled per user request, being possible coexisting several plugins running at the same time. During the application's startup, the plugin directory is scanned in order to identify the plugins that need to load and the configuration settings read to properly set up the properties of the plugins.

Figure 4.5 helps to understand the architecture followed by all developed plugins which are explained in detail in the implementation section of this document, including the request/response information data flow.

Plugin's calls are possible only through the available interfaces, including *plugin base interface*, *query interface*, *indexer interface*, *report interface*, among others. This assures a properly plugin load and that the methods are called correctly. These interfaces can be implemented by including the Dicoogle SDK as a dependency. Internally, although the existence of a well-defined independence between plugins, complex plugins may have their state shared with others. The state share is easily solved inside the *PluginSet* constructor which performs a properly instantiation of its child plugins and share any data structure required to achieve communication between its internal sub-components.

Another component specific of query plugins represented in Figure 4.5 is the query translator. When the running engine does not support the query syntax used by the core (Lucene syntax), it

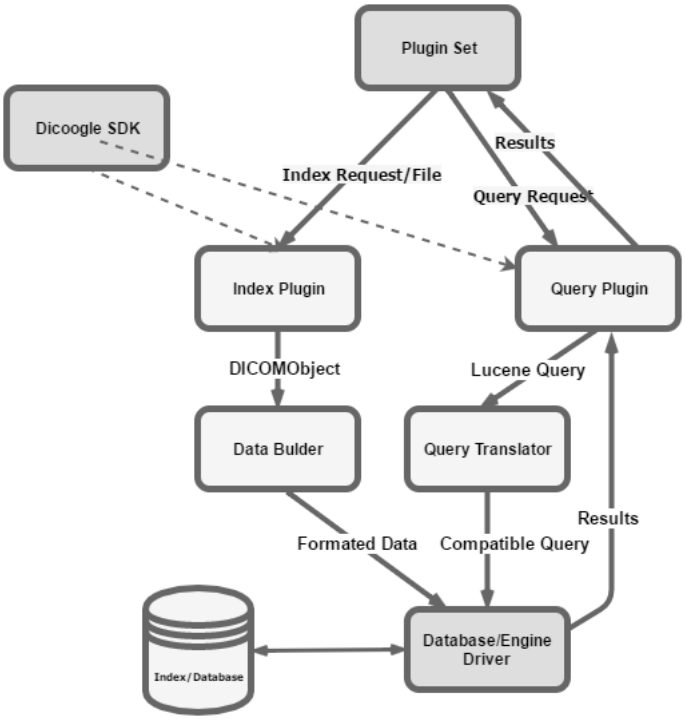

Figure 4.5: Plugin Software Architecture

performs a conversion from the text inserted by the user/service into the one understood by the search engine running. On the other side, Data Builder compose DICOM objects data into a format required by the indexing/database system engine, for instance, in a BSON object if operating with MongoDB.

Finally, the engine connector, that can be the usual JDBC, Apache Solr java API, among others and it is the component that performs the communication with the database/index.

4.3.3 DATA MODELS

The most important data models that need to be discussed are the ones used to translate the DICOM Object into a format that the indexing/query engines are able to understand. Both models can have different data, according to the running engine, which can follow a relational model or a non-relation(document oriented) structure, so it is possible to identify these two different data models.

The first one can be defined as a database schema, where the data is stored in tables. The second one is defined by documents, also know as JSON(or another similar format like a Java *HashMap<String,Object>*), and they can be nested or not.

The following sections described each of this models.

4.3.3.1 DATABASE MODEL

Next diagram present on Figure 4.6 describes a simplified database schema containing tables and relations used in the plugin that is implemented using a relational database.

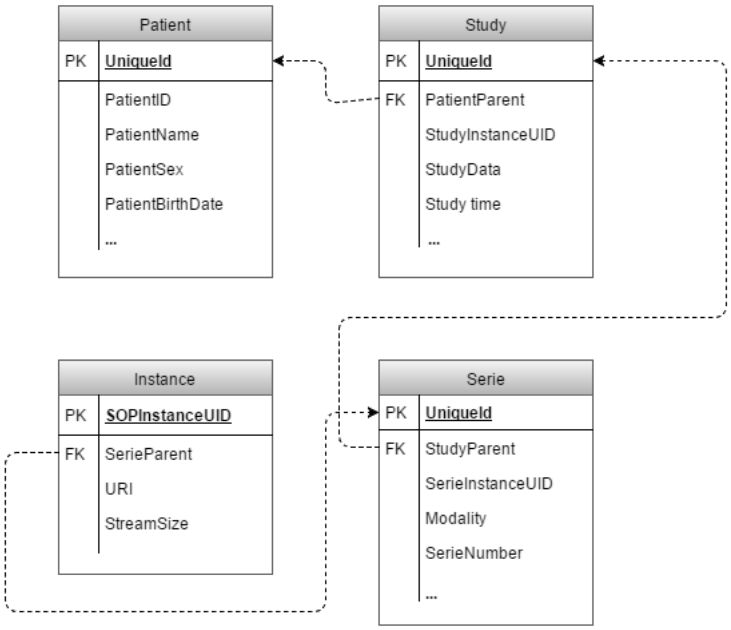

Figure 4.6: Database Schema

As described on the state of the art section, a DICOM file has a well-defined hierarchy. This hierarchy was applied in order to build the database schema.

On the defined model, an Image as a Serie parent (related through a foreign key), also a Serie belong to a Study and all studies are related to a Patient.

Because files often contain errors, is not possible to ensure that the *PatientID*, *StudyInstanceUID*, *SerieInstanceUID*, *SOPInstanceUID* are really unique keys, leading to the generation of a new unique identifier on each level(table). A single DICOM Object contains (according to DICOM standards) all this information levels.

4.3.3.2 DOCUMENT MODEL

More than one data structure can be used in the document model. Figure 4.7 represent on possible document used on the non-relational based solutions.

This kind of document contains only the most common DICOM elements from the hierarchy model. However, it can support many other, including private attributes. The existing elements depend on the set of tags defined by who is using the Dicoogle platform and depending on the service modality.

- "_index": "dicoogle",
- "_type": "filedicom",
- "_id": "AVT32mRQuPceXpezh4Fm",
- "_score": 1.0,
- "_source": {
  - "ModalitiesInStudy": null,
  - "PatientBirthDate": "19491210",
  - "PregnancyStatus": null,
  - "InstitutionDepartmentName": null,
  - "StudyDescription": "Thyroid Scan",
  - "PerformingPhysicianName": null,
  - "StudyInstanceUID": "1.3.12.2.1107.5.6.1.123.24412.0.231616614823159",
  - "SeriesInstanceUID": "1.3.12.2.1107.5.6.1.123.24412.0.231616727483060",
  - "SeriesDescription": "Thyroid Tc",
  - "StationName": "ECAM6653",
  - "PerformedLocation": null,
  - "Manufacturer": "SIEMENS NM",
  - "SeriesNumber": "1",
  - "SOPInstanceUID": "1.3.12.2.1107.5.6.1.123.24412.0.231015282982602",
  - "PatientPosition": null,
  - "OperatorName": null,
  - "InstitutionName": null,
  - "ManufacturerModelName": "IP2",
  - "Priority": null,
  - "StudyDate": "20010131",
  - "NumberOfStudyRelatedInstances": null,
  - "PatientName": "Thyroid^Uptake Syringe",
  - "uri": "file:/home/andre/Documents/tese/Deploy/dataset-ieeta/Thyroid%20Uptake%2(
    %201/ThyroidTc_1/IM-0001-0001.dcm",
  - "PatientAge": "051Y",
  - "StudyName": null,

Figure 4.7: JSON Schema Sample

As referred before, one of the big advantages of the non-relational databases is the possibility of having data stored on a non-fixed organization different from what happens in relational databases.

The above Document only contains one deep level. However, a DICOM Object can contain several nested objects with important data. For this reason, non-relation implementations have the possibility of indexing all fields contained in a DICOM object, solved with a recursive object iteration. This process will be further discussed with more details.

Figure 4.8 illustrates a possible nested document. On this unstructured data model, each document can be different from another within the same index, because the DICOM elements vary from file to file as already explained.

Figure 4.8: Sample Schema of a Nested JSON

CHAPTER 5

Database Design and Implementation

This chapter introduces the fundamental concepts and important implementation details regarding the developed index/query solutions for the existing Dicoogle's platform. The developed solutions involving different technologies will be fully described, presenting the performed optimizations on each one, including the applied configurations and some decisions taken that have influence in the overall performance.

5.1 relational database solution

Although the topic of this thesis refers to the study of No-SQL databases appliance on Dicoogle's platform over big data environment, relation databases also need to be evaluated to decide if this model can bring advantages to a final solution and even be used in a hybrid approach that can serve both identified PACS scenarios.

This section besides enumerate the existing advantages of this kind of databases, it also illustrates how the data model can be adapted with a XML configuration or through the changes on the Object/Relational Mapping (ORM) entities. The main steps of the index task, the process of query transformation and the way that results are build in order to be returned to the Dicoogle platform will be also described.

5.1.1 relation database based option

Almost every PACS solution relies on relation databases to support their archive systems, and for sure, this kind of databases bring several advantages with their appliance. One advantage example is the amount of redundant data on this systems, that can be lower than for example on a full-text engine. With the well-defined relations between, for instance, patients and studies, a given study only need to have information about the unique identifier (foreign key) of the correspondent patient parent, instead of having all fields from the patient level (patient name, birth data, patient sex, etc) on each

document. Other important advantages brought by this kind of engines and explained of the section 3 are also an important consideration factor for it development.

It is almost impossible to define a relational database schema that could fit with all the variety of combinations of DICOM elements. However, a regular search operations in traditional PACS, usually only require a pre-defined set of fields announced as DIM, establishing the information model that can be translated on a database schema.

With the development of a relational solution, some interesting points can be discovered related to performance issues and it also works as a point of comparison between the developed solutions, as well as check if this model can be used in a production PACS approach.

5.1.2 SCHEMA CONFIGURATION WITH XML

Dicoogle allows us to define the DICOM fields that we want to store. In order to keep a fully compatibility, the database schema can be defined through a XML file definition. For this feature, was used the *Commons Configuration* software library from Apache, that provides a generic configuration interface, which allows a Java application to read configuration data from different sources. The next XML sample, illustrates an example of how a field and its correspondent type can be defined by the user.

```
<database-schema>
    <tables>
        <table tableType="application">
            <name>Patient</name>
            <fields>
                <field>
                    <name>PatientID</name>
                    <type>VARCHAR(255)</type>
                    <isPK>true</isPK>
                    <isFK>false</isFK>
                </field>
                <field>
                    <name>PatientName</name>
                    <type>VARCHAR(255)</type>
                    <isPK>false</isPK>
                    <isFK>false</isFK>
                </field>
        ....
        ....
```

With this structure, a table name called `Patient` is declared containing the fields `PatientID`, with the data type `VARCHAR`, declared as a primary key. In order to build foreign keys, the tag `isFK` identifies the fields used for that purpose. After parsing, the SQL statements are formatted and executed to build the database schema on the running engine.

5.1.3 SCHEMA CONFIGURATION WITH ORM

After the implementation of the configuration described above, the resulting code that allows the fulfillment of the indexing tasks was rather verbose. In order to organize and simplify it, Hibernate software proved to be a good alternative. Hibernate ORM framework provides developers an easily way of writing applications with data persistence applied to relational databases (via JDBC).

Despite the need of recompilation to change an entity field, the scaling benefits exceed the XML configuration approach, including a powerful encapsulation, reduced programming overhead and, most important, it can be applied over several DBMS selected on a configuration file.

The following code illustrates how a *Hibernate* entity can be declared:

```
@Entity
@Table(name = "STUDY")
public class Study {

    @Id
    @Column(name = "id")
    private String uniqueID;

    @Column(name = "StudyInstanceUID")
    private String StudyInstanceUID ;

    @Column(name = "StudyID")
    private String StudyID;

    @Column(name = "StudyDate")
    private String StudyDate ;

    @Column(name = "StudyTime")
    private String StudyTime ;

  ....
```

This code stretch, after initialization and data fetch, creates a `Study` table, with the String attributes `id`, `StudyDate`, `StudyInstanceUID`, etc. The SQL statement generated is approximately: `create table STUDY (id VARCHAR(50) NOT NULL, StudyInstanceUID VARCHAR(50), StudyID VARCHAR(50) default NULL, ..., PRIMARY KEY (id));`.

5.1.4 INDEX OPERATION

On relational database based solution, index tasks are decomposed in three main operations: fetch the data, prepare and save it. For simplicity, the next description only has the steps regarding the Hibernate ORM usage.

Figure 5.1 demonstrates the main data-flow of this task.

The storage plugin, after an index task call, performs a directory scan in order to read the different file streams and send them over the indexer interface. If they belong to the DICOM format and after initialized and applied the user settings, the indexer plugin is prepared to receive this several streams

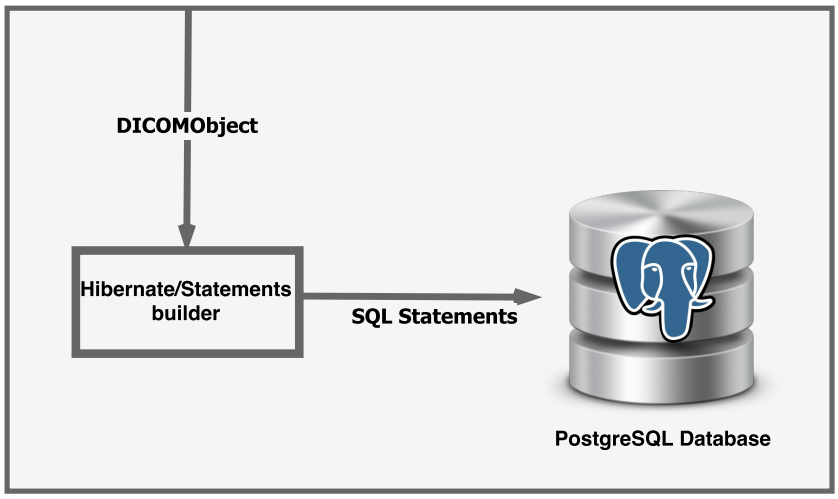

Figure 5.1: Relational Based Data Flow of Index Operation

with the DICOM objects.

For each DICOM file read, the data stream is analyzed to transform it into a data structure known as `DicomObject`. With `Hibernate` ORM usage, the data entities capable of storing the DIM attributes are instantiated and filled their properties with the data retrieved from the previously created structure.

These entities are the ones explained before, the Patient, Study, Series, and Instance. Some DICOM files can be corrupted and some relevant information can be missing, leading to errors while indexing, for instance, if a patient identifier is missing and that field was set as primary key, the studies, series, and instance will be lost because a primary key can not be `null`.

To avoid important data loss, the IDs are created by generating an `hash` from the `PatientField` and `PatientName` on the `Patient` level and with a similar strategy on the remaining entities. Auto-increment identifiers were excluded because they require an extra database query when inserting a foreign key on an entity. The `hash` strategy was possible because every DICOM stream, as explained before, contain the information about which Series it belongs, which Study and Patient allowing a single data access for every key to the `DicomObject` being available, for instance, the Study identifier is available when inserting a Series foreign key on the Series entity. With auto-increments, a query must be performed to get the auto-generated ids.

After having the entities filled with their properties, they are saved using the correspondent `Hibernate` method. As referred in the requirements section, storing operations have to rely on transactions. If a study is missing or some important field can not be read, the remaining entities from that file are not saved, to avoid erroneous relations.

The commit is performed when the number of objects reaches a given batch size, leading to an improved performance. The required memory depends on that size and after some small trials, the one with the best ratio memory/performance was chosen.

With the XML tags illustrated in sequel, the `Hibernate` tool can be configured to use a certain RDBMS, in this case, the PostgreSQL database has been used due to its advantages described on the state of art section of this thesis and due to some studies like the one from [73].

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
```

```
        <property
            name="hibernate.connection.url">jdbc:postgresql://localhost:5432/dicoogledb</property>
        <property name="hibernate.connection.password">dicoogle</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.search.default.directory_provider">filesystem</property>
        <property
            name="hibernate.search.default.indexBase">/home/andre/Documents/tese/Deploy/hibernateIndex</property>
        <property name="hibernate.jdbc.batch_size">50</property>
        <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
        <property name="show_sql">false</property>
        <property name="hbm2ddl.auto">create</property>
        <mapping class="dicoogle.ua.relationalpluginhibernate.entities.Patient"></mapping>
        <mapping class="dicoogle.ua.relationalpluginhibernate.entities.Study"></mapping>
        <mapping class="dicoogle.ua.relationalpluginhibernate.entities.Serie"></mapping>
        <mapping class="dicoogle.ua.relationalpluginhibernate.entities.Instance"></mapping>
    </session-factory>
</hibernate-configuration>
```

5.1.5 QUERY OPERATION

Query operation is the process through the user access to the desired information. On of the possible ways of calling this operation through the Dicoogle Web interface. The query is inserted and the components are displayed to show the retrieved results.

This operation can be decomposed into two different operations explained in the next subsections, the query transformation and the result set build.

Figure 5.2 helps to visualize the simplified requests/response flow.

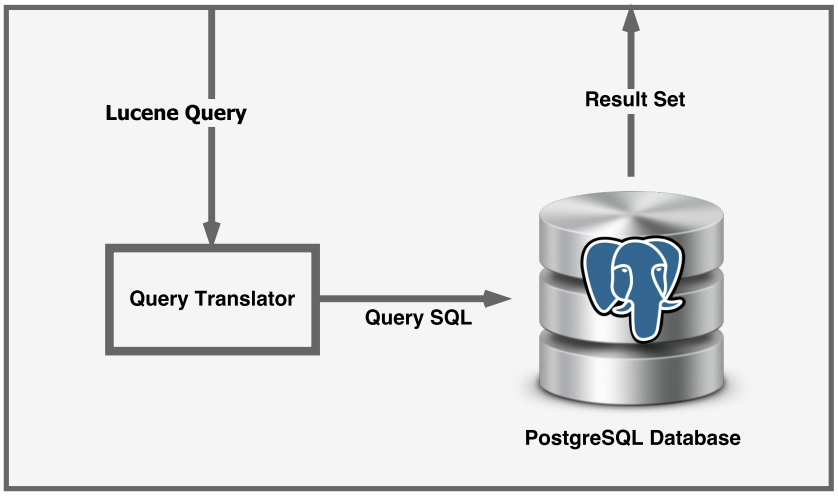

Figure 5.2: Relational Based Data Flow of Query Operation

5.1.5.1 QUERY TRANSFORMATION

Dicoogle only allows the query insertion using the Lucene query syntax. With the relational database implementation, a translation mechanism to SQL was necessary. After some research, no open source implementation was found, so it was fully implemented in the scope of this thesis.

A basic SQL statement is composed by three distinct clauses, the one starting by `SELECT`, other started with `FROM` and the last one started with `WHERE`.

On the `SELECT` clause, are indicated the fields that must be retrieved. Besides the query, Dicoogle query method also has an argument with the fields that should be retrieved from each document, this facilitates the creation of this clause. The `FROM` clause as explained in the next section, needs to retrieve all the hierarchy information, so it can be translated into the SQL statement: `PATIENT inner join STUDY on PATIENT.id = STUDY.parent_id inner join SERIE on STUDY.id = SERIE.parent_id inner join INSTANCE on SERIE.id = INSTANCE.parent_id `, aggregating the data from all tables based on the existing relations. Finally, the "WHERE" clause is inferred through the analysis of the query introduced by the user, this because a Lucene query syntax usually has the structure `FieldName:Constant` allowing to build the `WHERE` from it parse, resulting in `FieldName = Constant`.

Other important features to enhance the available query options were developed, including the `AND`, `OR`, `NOT` operators from compose queries. In order to perform wild card queries, and offer better results, the `LIKE` operator was applied. This operator allows searching for a specified pattern in a column (field). Composed queries features are available, also through the query analysis.

For a better perception, Figure 5.3 contains a translation sample.

When the SQL statement is prepared, it is executed using the JDBC. The retrieving steps are explained in the next section.

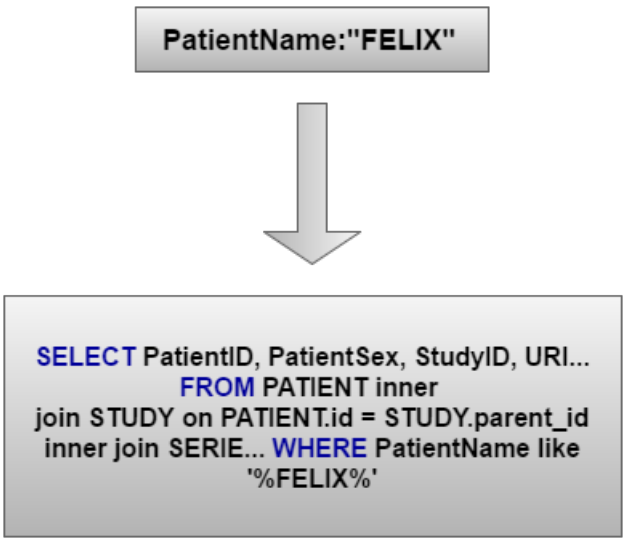

Figure 5.3: Sample SQL Query Translation

5.1.5.2 RESULTS BUILD

A result table called "result set" is returned after the execution of a `SELECT` query, retrieved by the DBMS and matching the requested data. Those results need to be converted into a data structure that Dicoogle can understand and available on the interfaces provided by the SDK.

The returning object is a list containing several rows from the result set where each row is stored into a `HashMap` alike data structure. The `HashMap` keys are the tags alias from the data fields, and the values are the actual constants matching that fields.

To avoid large memory usage, a cursor object called `ShardResultStream` is returned, after the response from the database engine, which allows Dicoogle to get as many results as it needs with the right memory control. This can be performed trough the use of cursors available on the relational database engine. The number of results pre-fetch in memory can be configured by setting up the bath size variable.

After an iteration, the data is analyzed by the Dicoogle's core, building the data structures that contain all the meta-data for the graphical interface or DICOM service support or perform any other task with the retrieved document. When the pre-fetched results are all processed, PostgreSQL performs a query to get more results.

5.1.6 PERFORMED OPTIMIZATIONS

To obtain the best possible result from the relation database engine, some crucial configurations were applied. PostgreSQL settings can be changed by manipulating its configuration files. Several small tests (over small datasets) were performed in order to obtain the best possible performance with the available resources.

The first parameter that needs to be configured according to the running server specifications is the shared_buffers property. This parameter tells PostgreSQL how much memory it can use to data caching. The value 250 MB was defined, although the recommended is 1/4 of the systems memory, in order to keep the fairness criteria with the other tested solutions.

The next changed variable was the effective_cache_size that sets up the how much memory can be used for disk caching the database, for that PostgreSQL query planner expects what it is possible to fit in RAM and what it can not, being 1GB the chosen value.

The last changed property was the `autovacuum` that was disabled, because when loading large amounts of data (bulk operations), it can seriously affect the performance on the maintenance chores that it performs.

Besides this configuration updates, in order to improve the query response time, several indexes were created, including on the most frequent searched fields, like for instance, in the fields `PatientAge`, `Modality`, `StudyInstanceUID`, etc. Although they delay the index operation, they bring performance improvement on the query operation.

5.2 NON-RELATIONAL DATABASE SOLUTION

In accordance with the relational database based solution section, it will explain the reasons for the development of a non-relation database based solution. After that, some important implementation details regarding main operations of the index operation, the query transformation, how the results are build in order to successfully perform the retrieval task and finally the performed optimizations will be presented. It will be also introduced how the research PACS use case scenario was achieved in the following implementations, indexing and allowing queries over all the fields existent in a DICOM file.

5.2.1 NON-RELATIONAL DATABASE BASED OPTION

DICOM objects contains data and metadata that can fit into a non-tabular structure. Non-relational databases are prepared to store, retrieve, and managing this kind of information format.

As already explained, document-oriented databases are very different from the traditional relational databases, with a very high growth rate. When a certain field needs changes on its properties, instead of changing the database schema, they can adapt in a very simple way. Given the big data context, this technologies progress, the overall performance, and validation of the brought benefits, they are crucial for this work. MongoDB was the selected NoSQL database due to the advantages of this engine already assessed in the related work section of Chapter 4.

5.2.2 INDEX OPERATION

Some operations are quite similar to the ones previously explained on the relational database based solution. However, there are several important differences that have to be taken into consideration.

Figure 5.4 clarify the main differences in the data flow followed by this solution.

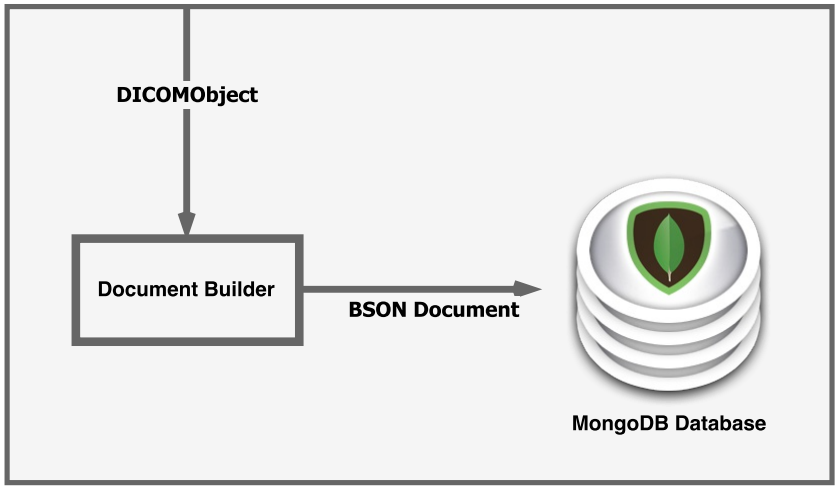

Figure 5.4: Non-Relational Based Data Flow of Index Operation

From the read of a DICOM file to the creation of the `DicomObject` structure the data flow is exactly the same, but after that, two different operation modes can be executed in order to perform the data indexing. The operation selection is based on the configuration file, where a flag designated `DIM` can be `true` or `false`, according to the goals that the user is using the PACS archive.

The first one does not index all the fields from a DICOM file, but only the chosen fields from the already explained DIM from the relational model, to be used in a traditional/institutional PACS. To build the required MongoDB document DBMS, a BSON document available through the Java API has been used. Dicoogle already has an implementation of a data structure called `TagsStruct` with a method that allows the retrieving of several tags that are configured as objects belonging to the DIM.

With the `DicomObject` and the referred `TagsStruct`, and iterating over all the tags alias, it is possible to fill the document. At the end of this building process, it needs to be saved in the database.

The plugin configuration file also contains the database name and database collection where the documents storage will occur. With an already open session from the indexer plugin initialization, a API call is made to the configured collection to insert the document.

On the second operation mode, the main goal is to index all the data and meta-data present on the DICOM file focused mainly for research purposes in the PACS use case scenarios. A recursive function has been developed to exploit the main DICOM object, and it allows to unmarshalling the inner objects and get access to all the fields, in order to create a nested BSON, representing the same hierarchy from the original DICOM object. The data types from each field are detected for a proper store.

Table 5.1 illustrates the several data types that an object can contain. The saving steps are equal to the ones performed in the DIM operation.

String
Integer
Integer []
Double
Double []
DICOMObject

Table 5.1: DICOM Object Data Types

5.2.3 QUERY OPERATION

This task goal is the same as the one in the relational database based solution, however, there are several important differences in the way that the response is build. The query method of this solution is also composed of two main operations: query translation and the results build. One the first sub-section the query transformation will be explained, which is the translation process from the Lucene syntax query to the one understand by the MongoDB DBMS.

All the included features will be detailed described. Finally follows a brief explanation of how the resulting documents are retrieved from the database store to the Dicoogle's core application, called as result build operation. Figure 5.5 describes the way that requests are processed.

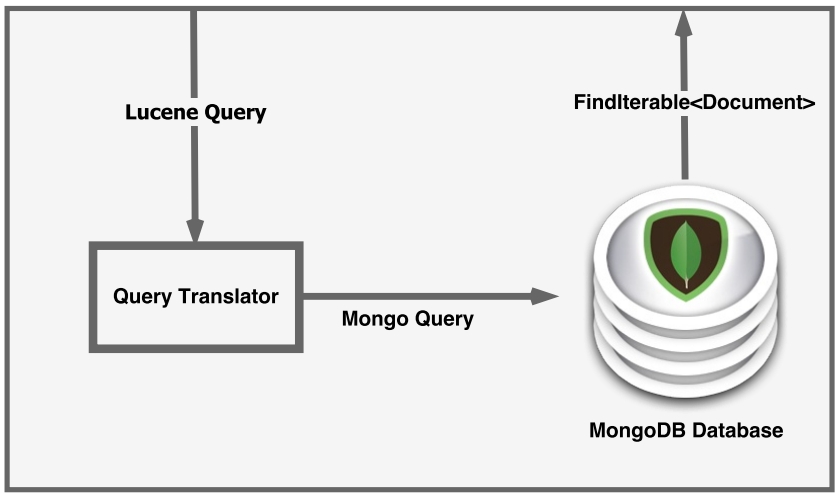

Figure 5.5: Non-Relational Based Data Flow of Query Operation

5.2.3.1 QUERY TRANSFORMATION

Lucene and MongoDB have very different query syntax. In order to keep a full compatibility with the current platform, a complete translation was developed.

The search method of the MongoDB Java API (`Find`) receives a Java object called `DBObject` and this object has a constructor that receives a string with the desired query. The first basic translation is the one that selects all the existing documents, expressed on Lucene by the query: "*:*" and can be translated on an empty `DBObject`. On the remaining queries, it is performed a parse char by char to get information about the query goals. For instance, if a ":" is detected, the concatenation of chars until it occurs is the field component of the query. Also, the char "[" gives information about a range query, and the string "TO" helps to separate the range constants. "NOT" filter is also detected in order no proceed to its translation. Another important goal detection is the operators `AND/OR` detected by the same process. The presence of quotation marks allows the identification of the desired data types.

After identifying the query goals, it is necessary to translate into one that MongoDB can understand. Three different query types are translated, based on the knowledge previously acquired with the parsing: the range query, value query and the regular expression query. Range query uses the operators "$gt" which selects the documents where the value of the field is greater than, "$lt" selection the ones less than, and if it is an inclusive query, "$gte" and "$lte" are used to get also the equivalent documents. The value query performs direct mapping with exception of "NOT" queries that require the "$ne" operator to select the documents that do not contain the specified value. Finally, regular expression queries use the "$regex" MongoDB operator.

Figure 5.6 illustrates a possible translation.

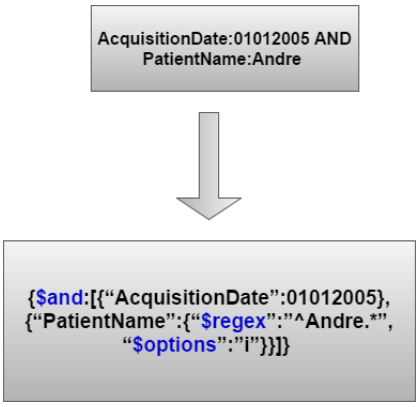

Figure 5.6: Sample MongoDB Query Translation

5.2.3.2 RESULTS BUILD

After executing the MongoDB search method over the database containing the indexed documents, the returning data needs to be processed into a structure that Dicoogle's core can understand. Results are retrieved on an Object called `FindIterable<Document>` that is an extension of the Iterable class containing a way of access to the resulting documents. The same iterative strategy from the relational based solution was applied. The main difference is that MongoDB already sets a cursor on the iterable object, caching and fetching data when needed.

Distinct from the previous `HashMap` to `HashMap` translation, on the non-relational solution, a very similar `Document` to `HashMap` is performed for each request.

5.2.4 PERFORMED OPTIMIZATIONS

Several configurations were changed in order to tune up MongoDB performance. Following the engine official website (www.mongodb.com) and performing small tests, it was possible to improve the solution performance on query and index operations.

MongoDB 3.0 release provides a new storage engine called `WiredTiger` with new features that include, for instance, compression for all collections and indexes minimizing storage use by increasing CPU load. `WiredTiger` use a minimum of 1GB RAM, so this configuration was not changed although it can improve significantly the overall performance if the right resources are available.

The first changed variable was the `journalCommitInterval`, which defines the writing frequency in the hard disk that and has a default value of 100ms, by changing it to 300ms some resources are saved. Disabling the BSON validation using the `objcheck` also improves the index speed. This can be done because the correctness of the BSON documents is ensured when the indexing documents are built inside the index plugin.

On big data environments, disabling profiling (no query logging) also brings several performance benefits. This can be achieved by changing the profile variable to 0.

Apart from changing configurations, query performance was also improved by setting up indexes on the same fields as the ones from the relation database based solution, because scanning an index is much faster than scanning a collection and it influence on the index time is minor.

5.3 ELASTICSEARCH BASED SOLUTION

ElasticSearch, as described on the state of art section, is one of the most popular search engines based on Lucene. In this section, it will be explained why an ElasticSearch based solution was developed, the main steps involved in the index and query tasks, including some important decisions taken. This solution is also capable of index both PACS scenarios with some present differences. Finally, the main optimizations performed in order to obtain a better performance.

5.3.1 ELASTICSEARCH BASED OPTION

Although Elasticsearch is a search engine based on Lucene, it provides a distributed, multitenant-capable full-text search engine with a lot of extra-features, trying to have all its features available through a Java API. Caching mechanisms, different ranking algorithms, several query handlers and so on are some extra features that can bring value as an alternative to the current Lucene based solution.

This engine also has a lot of compatible tools that were already very tested and widely applied, for instance, Kibana, which allows explore and visualize data and can be useful for research purposes.

Because it is open source under the terms of the Apache License, and because of the above illustrated features, an index and query plugins using this engine were fully developed and tested in order to take conclusions about its performance in the proposed big data environment.

5.3.2 INDEX OPERATION

ElasticSearch engine provides an index API for adding or updating JSON documents on a specific index, making it searchable. Like explained on the index operation sections of the previous solutions, the data flow until the indexer method on the indexer plugin is the same. After receiving a call with the `StorageInputStream`, and converted into a `DICOMObject` data structure with all the DICOM elements from a certain file, the process of transformation in a document compatible with ElasticSearch is performed, followed by the commit to the index.

Figure 5.7 is a schema containing the necessary steps for this index task. The index also needs to be previously created on a node inside a cluster. All these names are parameterized inside it configuration file.

This solution also has the possibility of all fields indexing or only the ones from DIM. These two different data models are build using the same strategy from the MongoDB-based solution, with the resulting data structure being the main difference. A `HashMap<String, Object>` was used to perform the data index, because it is possible to directly index it on ElasticSearch.

Despite a `DicomObject` can be saved in an Object (declared in the ElasticSeach mapping), the same recursive algorithm has been applied to iterate over nested `DicomObjects` in order to keep the same data format on all developed solutions and with the existing one based on Apache Lucene.

After created a client connection over a configured node, the ElasticSearch server is ready for receiving the index requests.

ElasticSearch version 2.3 supports a Bulk operation where it is possible to perform many index/delete operations with a single API call. This leads to an increased indexing speed. Several bulk

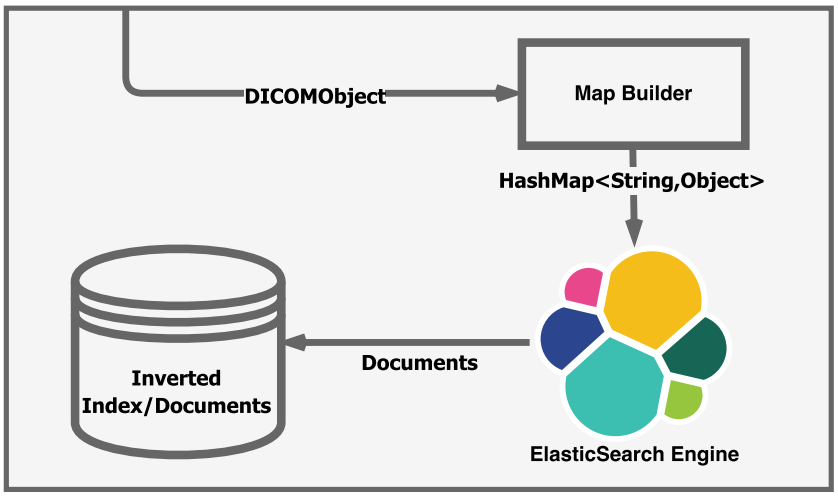

Figure 5.7: ElasticSearch Based Data Flow of Index Operation

sizes were tested to get the best ratio between the amount of memory usage (number of documents saved in ram before index flush) and the index speed.

Regarding document identifiers, as recommended by the ElasticSearch community of developers, they were automatically generated.

5.3.3 QUERY OPERATION

Based on Apache Lucene, ElasticSearch does not require a translation from the current supported Dicoogle query syntax to the one used by this engine. ElasticSearch Client from the Java API provides a `prepareSearch` method, which allows setting up all required parameters necessary to perform a query.

The first defined setting is the index name that corresponds to the previously created index, followed by the search type. After several trials, the one that fits all required features without having a huge impact on the performance is the `SCAN` type. This search type allows very efficient scrolling through large result sets. The next parameter is the query, obtained through the SDK interface. It is encapsulated on a `QueryBuilders` Object also specifying the query type, that was defined as a "matchQuery" type, which accepts text values, numerical or even dates. This kind of query analyzes the given fields and their data types, producing the final query.

After defining the number of returning documents (rows number) a scroll parameter tells the search engine to perform a scroll request, retrieving on the response a cursor identifier to allow a deep search over the results.

On the results retrieving, the process is similar to the described on the relational and non-relational based solutions, also following an iterative approach. When the buffer containing the search results is empty, a new ElasticSearch call is performed, given as argument the cursor identifier retrieved on the last response, resulting in a deep search that is more efficient, avoiding the manual set up of positions where the engine should start retrieving.

Next diagram 5.8 resumes the explained operations.

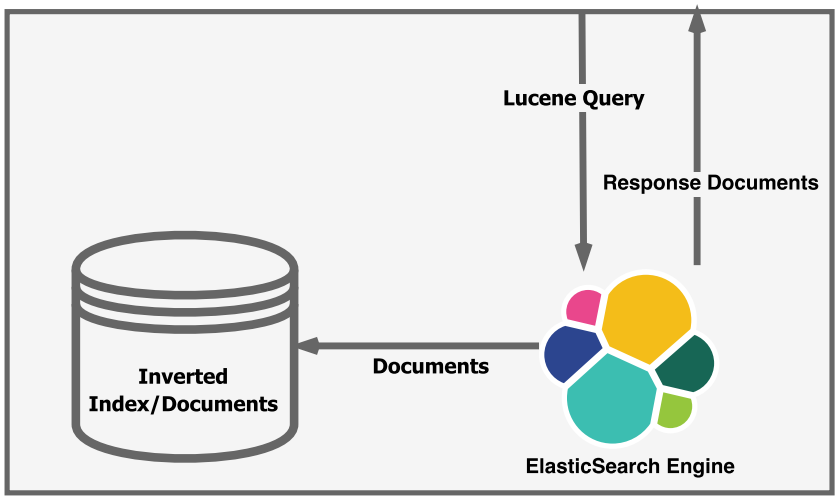

Figure 5.8: ElasticSearch Based Data Flow of Query Operation

5.3.4 PERFORMED OPTIMIZATIONS

Architecture and system configuration are factors that can affect database performance and responsiveness. For a better performance on this big data environment, several configurations were changed on the associated operative system.

First, the swapping memory for the Elasticsearch process was disabled, because swapping is very bad for performance and for node stability according to the ElasticSearch official Website.

The heap size configured for Elasticsearch was 1GB. The number of open files descriptors on the machine was changed to 64k as recommended. On the ElasticSearch engine, the mapping of the Dicoogle index, suffer some design alterations while comparing to the default. The `norms.enabled` variable was changed to false on certain non-relevant fields because it calculates a number to represent the relative field length and other values necessary to compute a score that will not be needed.

Finally, almost all fields are single valued and do not need to be analyzed (applying stop-words and tokenization) so they have a property designated "not_analyzed" to prevent ElasticSearch to perform heavy operations over fields that probably will not be searched.

5.4 SOLR BASED SOLUTION

Apache Solr, like ElasticSearch, is widely applied in many indexing/query systems. Following the same structure as the previous section, the reason for the development and test of a Solr based solution will be explained, followed by a description of how the index and query tasks are performed on the sub-plugins, highlighting why certain choices were made. Lastly, some relevant implemented optimizations are also described.

5.4.1 SOLR BASED OPTION

Apache Solr is a full-text search engine also based on Apache Lucene. This engine includes several new features that bring highly reliability, scalability and fault tolerant, providing distributed indexing, replication, and load-balanced querying.

In order to study its performance and these new features, it seemed a good option also to explore and develop a full Solr based solution. SolrJ is a Java client API which allows the access to almost all methods. It offers an interface to add, update, and query the created Solr index. The `HttpSolrServer` also has a good interface to give feedback about the cores state, the current indexes, memory usage, perform direct queries among other important features.

5.4.2 INDEX OPERATION

In this task, getting the input stream from the Dicoogle core through the storage plugin, forming the `DICOMObject` and getting the `HashMap` with the DICOM elements are operations performed in the same way that the ones described in the ElasticSearch solution. After that point, there are some relevant differences in the operations necessary to execute in order to perform an indexing task and synthesized in Figure 5.9.

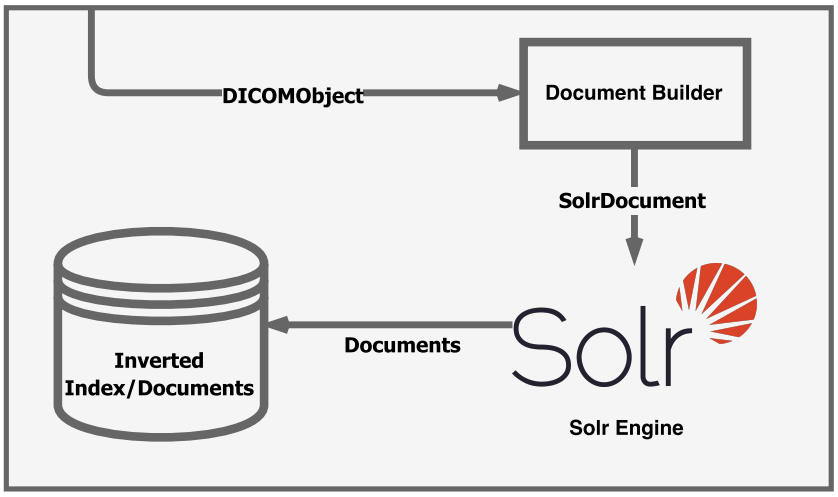

Figure 5.9: Solr Based Data Flow of Index Operation

SolrJ has a data structure called `SolrInputDocument` and it was used to save data into a Solr Index. The index creation is based on a defined configuration file for the plugin settings. After the appliance of other configurations like core name, address, port number, etc, connecting to the server requires instantiating a `HttpSolrClient`, that has available an interface for adding documents, commit to the server, among other relevant operations.

Solr based solution also contains two different ways of indexing a document: DIM and all fields for the traditional and research PACS scenarios respectively. This two operations modes are also implemented and configured in the same way as already explained on the previous solutions. These documents need to follow a defined structure on a schema file as described in the optimization section, however, the fields can be empty.

The document's unique identifiers are generated automatically by the Solr Engine, as recommended by the developers. On the predefined fields operation (DIM), the field names are defined in the XML Schema file that is associated with the Dicoogle index, so that the analysis steps such as tokenization and stop words appliance can be chosen singly.

Fields that are not explicitly defined in the Schema (the ones remaining in the all fields operation) will be mapped to a dynamic field definition. After completing the document creation, it is added to the batch for indexing. Several batch sizes were also tested in order to obtain the best trade off between memory and indexing speed.

5.4.3 QUERY OPERATION

Solr is also based on Lucene, so it does not require a query translation. The communication with the Dicoogle's core follows the same strategy from the already described solutions and only the most important differences will be explained.

To query a Solr index from the SolrJ API, it is used a `QuerySolr` Object, which allows defining several important properties containing the query options. Several handlers can be configured to process different kinds of query requests and this object allows to choose which one will be selected. The query `String` requested is also inserted, combined with the information needed to sort the results. The applied sorting parameters will be addressed on the following optimization section.

Finally, a `setRows` method is invoked to define the batch size that defines the number of results. On the results retrieving, a cursoring technique is used and is very similar to a deep paginating approach. `SolrDocumentList` is the data structure present in a query response, which allows access to the result set documents. Together with the `SolrDocumentList`, a cursor mark is also retrieved in the response to be used on the next document request. As soon as the batch is empty and the Dicoogle requires more results passing the last saved cursor mark which will be updated.

Figure 5.10 helps to visualize the explained operation.

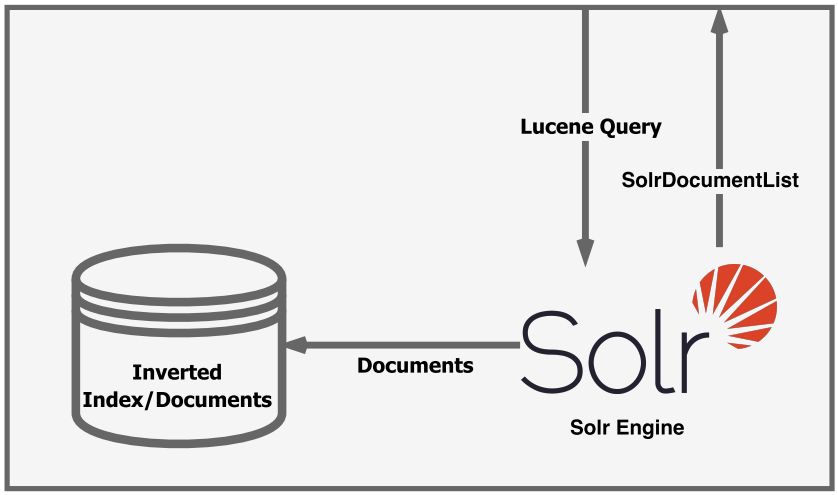

Figure 5.10: Solr Based Data Flow of Query Operation

5.4.4 PERFORMED OPTIMIZATIONS

Besides the usual changes on the memory settings and on the Java virtual machine configuration, the main performed optimization were done by configuration of engine parameters in the `solrconfig.xml` and `schema.xml` files.

On the Solr configuration file, it is possible to define the index/query request handlers parameters related to replication, caching settings, the schema file used, among others.

The first property changed was the `filterCache`, which is an `autowarm` cache populated with the most recent accessed items, used by the `SolrIndexSearcher` for documents that match a query and are available on the `queryResultCache`. This `queryResultCache` has the maximum RAM that the cache can occupy, both changed to twice the default value because it is predefined with low values.

On the schema file, the fields belonging to the DIM were configured according to its properties. On the other side, for the all fields implementation it was necessary to use a dynamic mapping available on the new Solr release because it is not possible to know all the existing fields before all DICOM being processed.

The used fields types are illustrated on table 5.2 with the correspondent description, having information about the used analyzer, the Solr type, and other relevant properties. The full mapping is available in appendix 2 of this thesis.

Alias	Solr Data Type	Properties
boolean	solr.BoolField	Single value and not analysed
string	solr.StrField	Single value and not analysed
strings	solr.StrField	Single value analysed
stringss	solr.StrField	Multi value and not analysed
stringsss	solr.StrField	Multi value analysed
long	solr.TrieLongField	Single value and not analysed
tlongs	solr.TrieLongField	Multi value and not analysed
double	solr.TrieDoubleField	Single value and not analysed
tdoubles	solr.TrieDoubleField	Multi value and not analysed
tdate	solr.TrieDateField	Single value and not analysed
tdates	solr.TrieDateField	Multi value and not analysed

Table 5.2: Defined Field Types on Solr Schema

An analyzer examines the text of the fields and generates a stream of tokens. The default analyzer was changed because it performs operations that are unnecessary and heavy computationally.

The one used apply stop words filter, a lower case filter and a simple tokenization algorithm. After defining the field types, the field tags that belong to a document were declared and some important variables overriding the defaults and working as flags were defined. This allows Solr engine to have knowledge about what values it will expect on each field and what it will do to them. The most important are described in the following Table 5.3 together with a brief description of their function.

The Solr engine requires the appliance of a sort strategy for iterating over a large result set (different from ElasticSearch). Resulting from several trials and research, the parameters that have the best results were the document `id` with a descending order.

Variable	Description
type	Associate to the field the data type explained above.
indexed	Indicates if the field be added to the inverted index in order to be searchable
stored	Configures if the field is stored in order to allows it's retrieve
required	If it is possible or not add a document to the index without the correspondent field

Table 5.3: Important Variables to Configure in Solr Schema

CHAPTER 6

Validation / Results

To find the database solution that has the best performance against the current based in the library Apache Lucene, many comparing tests had to be carried out. First in this chapter, the test conditions of the developed solutions are presented. The developed plugins need to follow some strict rules in order to assure the maximum fairness among them. For example, they all should have available the same amount of RAM. After that, it is possible to visualize on a graphical format, the obtained results on several tasks, including the indexing and query times when dealing with different volumes of data (distinct datasets) and the amount of disk space used by the indexes/databases produced by the engines. With the exception of the relational database based solution, each of these results can be split into two main groups, the ones implementing the DIM strategy used in the traditional PACS and the ones implementing an all fields approach applied in research PACS. Finally, it is proposed a final solution that we consider to be the best for replacing the current solution in our PACS.

6.1 Test environment conditions and assumptions

The trials with the distinct database implementations were executed in a server running Ubuntu 14.04.4 LTS with the following specifications:

RAM	10GB
CPU	Intel(R) Xeon(R) CPU X5650, 2.67GHz, 2 Cores
DISK	HDD 1TB

Table 6.1: Server Specifications

To ensure a maximum fairness, the following conditions were ensured in all experiments:

- Same system load, including an equal number of processes within what is possible to control.
- Same amount of RAM available to the engines.
- Same available heap size and cache size.
- No replication strategies because Lucene-based solution does not allow it.

- Equal datasets with the same DICOM files.

Table 6.2 contains information about the datasets used to perform the trials.

Number of DICOM Files	**Required Storage Space**
863	651MB
147859	56GB
4295069	864GB
7524561	1.9TB

Table 6.2: Information about the Datasets

6.2 TEST ALGORITHMS

Three main testing algorithms were developed in order to automatize the execution of the several tasks. The first one is responsible for setting up the environment, the second one for testing the index method and the last one for testing the query. These scripts were developed using Python language, because although the Dicoogle's Java implementation, it has a python wrapper which allows the call of some core methods from its services including the index and query.

The script used to set up the environment has a method that, after receiving an alias with the desired execution solution, creates the necessary directory tree, the symbolic link to the executable `jar` and also has a method for clean up to be called after the execution. The indexing script has information about the data directories of each solution and iterates over the datasets, passing the directory's location that will be scanned to the index method. The task time is saved into a file and the resulting index data directory is copied to a different location in order to perform queries over it later. The engines start and stop are also called before and after the method call. The last one, besides some similar tasks with the indexing script including the start and stop of the engines, it is responsible for copying the data directories to the correct location specified by the engines configuration files, calling the query method and finally store the operation metadata, including the elapsed time and the number of retrieved results.

6.3 DIM INDEXING SCENARIO

Nowadays, already were performed a huge amount of studies comparing NoSQL vs SQL databases and even comparing with engines based on the file system[74]. However, they were made using different data models, with different data, and in a distinct environment.

In this section, it will be presented the results from the five solutions (four developed plus Lucene based solution) on the performance of several tasks: index, query and also comparing the index sizes.

This DIM data model takes advantage of the hierarchy existent in the DICOM files, in order to build the relational model as explained before, and compares it with other solutions using other Engines/Java APIs to evaluate if it can bring advantages to the Dicoogle's platform.

Despite the fact that they are not using all the existing DICOM elements on a file, they can be very useful in several scenarios, including a full compatibility with the Dicoogle's web interface and traditional PACS appliances where the user knows which fields he needs to work with, saving this way some important resources.

-Index Operation:

Graphic 6.1 illustrate the obtained results of the index task over four different datasets.

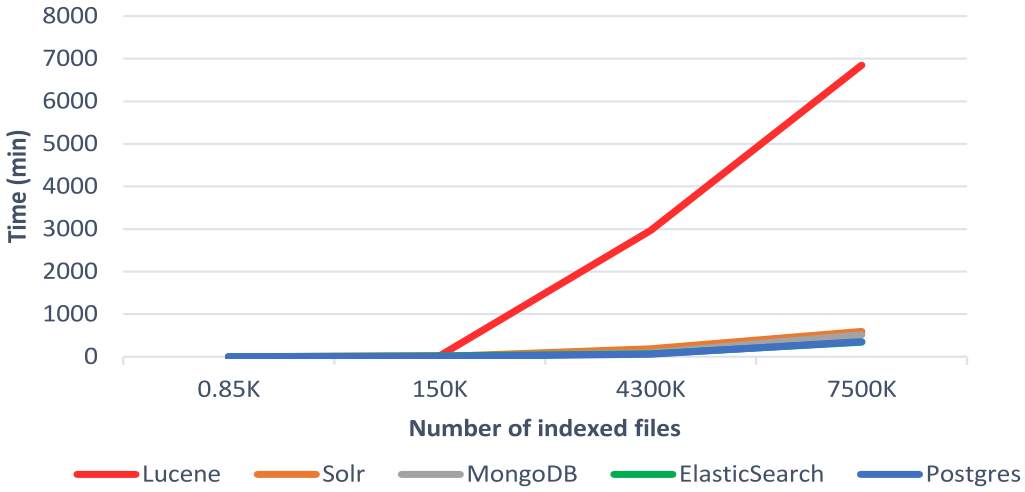

Figure 6.1: DIM Indexing Time of all Solutions

On the `Y` axis it is represented the time in minutes and on the `X` axis the number of files that each dataset has. The smallest dataset contains 850 DICOM files and the biggest one contains approximately 7.5 Million DICOM files.

On the first two datasets, it is possible to verify that the performance of the different solutions its quite similar, however, when indexing datasets which are near to the "big data environments", Apache Lucene based solution demonstrates a very poor performance. For instance, the dataset with 7.5 Million DICOM files takes more that 150 hours(6.25 days) to fully perform this task.

For a better visual perception of the results regarding the results obtained by the four developed solutions, Graphic 6.2 presents the results without Lucene solution. All of them got an indexing performance that can be considered good giving the big number required operations. The relational database and the ElasticSearch based have the best results, being the ElasticSearch based a little faster on the biggest dataset but significantly slower in the smaller ones.

As already explained, the required time to perform an indexing task usually is not a critical performance metric because it can be created in advance. However in medical imaging scenarios, with enormous data volumes generated every day, it slowness can obstruct the normal workflow of physicians, researchers, etc.

When looking towards Lucene based solution, this plugin shows performance degradation on a higher pronounced level when the dataset increases significantly. Besides some implementations faults, Apache Lucene library also has some issues that can lead to this poor performance. The existing solutions use Apache Lucene version 3.5.0 while version 6.0.0 was already released with significant improvements, which include new features and capabilities added to the streaming API, uses a minimum

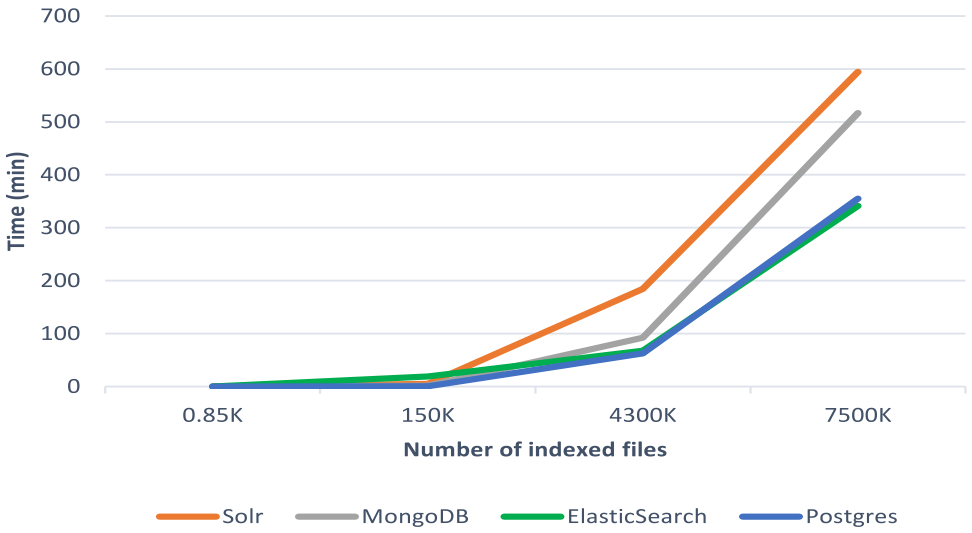

Figure 6.2: DIM Indexing Time of the Developed Solutions

Java 8 version and takes advantage of the new available methods, new data types instead of the older ones `numeric fields`, etc.

With no direct cache mechanisms, Java writing operations have bigger overhead while comparing to the techniques applied by the used Engines[75]. With caching mechanisms and bulk operations, the developed four solutions had quite similar performance. Relational database plugin achieved a good performance resulting from the Hibernate concurrent implementation, inserting several data rows with a well-implemented thread pool. ElasticSeach and Solr besides cache mechanisms also use recently released Lucene versions, improving this way their performance. Batch indexing (many documents indexed within the same call) were also used to have results significantly betters [76]. On top of this, unlike the developed plugins, Lucene only runs on a single node, making impossible to take advantage from a distributed model.

-Size of index on disk:

The data belonging to the DICOM files is stored in distinct means, for instance, the relational model stores into tables, the text-based solutions stores it in inverted index data structures and the non-relational based solution stores into several series of BSON files, leading to differences in the necessary disk space to store a certain index.

The results presented on Graphic 6.3 shows the occupied space for each solution on the different datasets. Note that these results represent the required space for all the files present in the data folder of each engine, including the extra database files from the running database engine, for example, the resulting logs and all the extra meta-data.

The data folder from the relational database solution occupies together with the non-relational based more space on the first datasets. This happens because of the database files as explained above. When the number of files starts scaling database solutions start to get closer with the full-text search engines. On the relational model, the nonexistence of data replication resulting from data tables relations saves more disk space. MongoDB last releases use a new storage engine called `WiredTiger` which performs data compression, requiring this way less disk space. Apache Lucene based solution

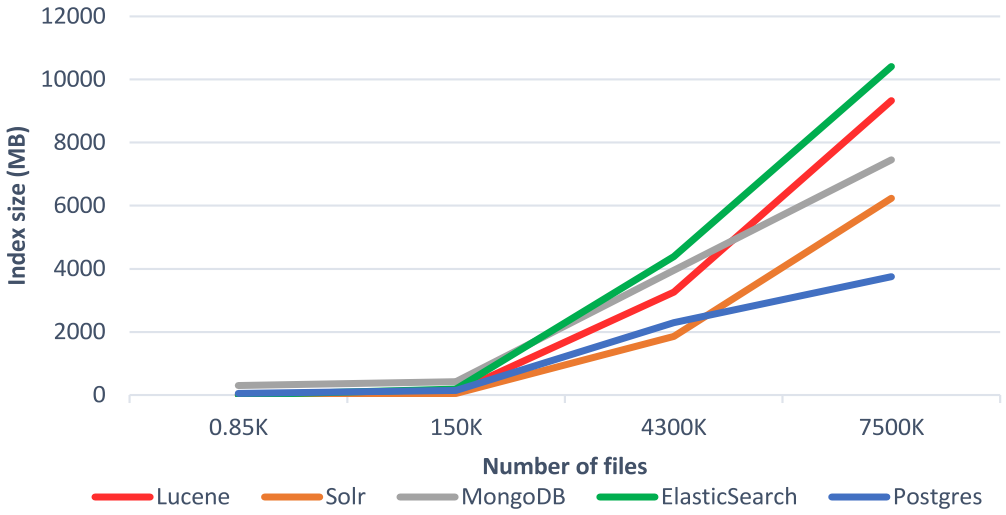

Figure 6.3: Index Size of all Solutions in the DIM Indexing Strategy

is currently saving Floats, Integers, and Longs as Strings, requiring this way more space due to it encoding.

Solr has the capability of saving this data types in their original format, however, its schema is not mutable, meaning that if a given field is saved first as String but after it has a Float value, it has to be saved in the String format. On ElasticSearch documents can have different types on each field, when this happens, the new property will be added automatically to the mapping definitions. The required space by the text-based solutions can vary significantly with the dataset, if in these ones they require more space, it can not always be true.

-Query Results:

The measures taken regarding query performance are divided into four distinct sections according to the size of datasets tested. For each one, several queries were performed with a different number of returning results, representing a percentage of the total number of documents. This technique tests deeply the solutions retrieving mechanisms and checks their behavior over different scenarios because it has queries from less than 1% hits to queries with 100%.

The service used invokes the query method passing the desired query and the fields that are exported into a CSV file, iterating over each results batch and writing them into a file. This strategy avoids memory overflow because on big datasets some queries can have millions of results.

Table 6.3 represents the queries performed over the smallest dataset.

Query	Number of Results
Modality:NM	12
Modality:XA	15
Modality:PT	244
Modality:MR	562
SOPInstanceUID:*	863

Table 6.3: Queries Used over the Dataset with 863 DICOM Files

The graphic 6.4 represents the results obtained while running the presented queries over a dataset containing 863 DICOM files. On the `Y` axis it is represented the query operation time and on the `X` axis the number of returning results from the correspondent query.

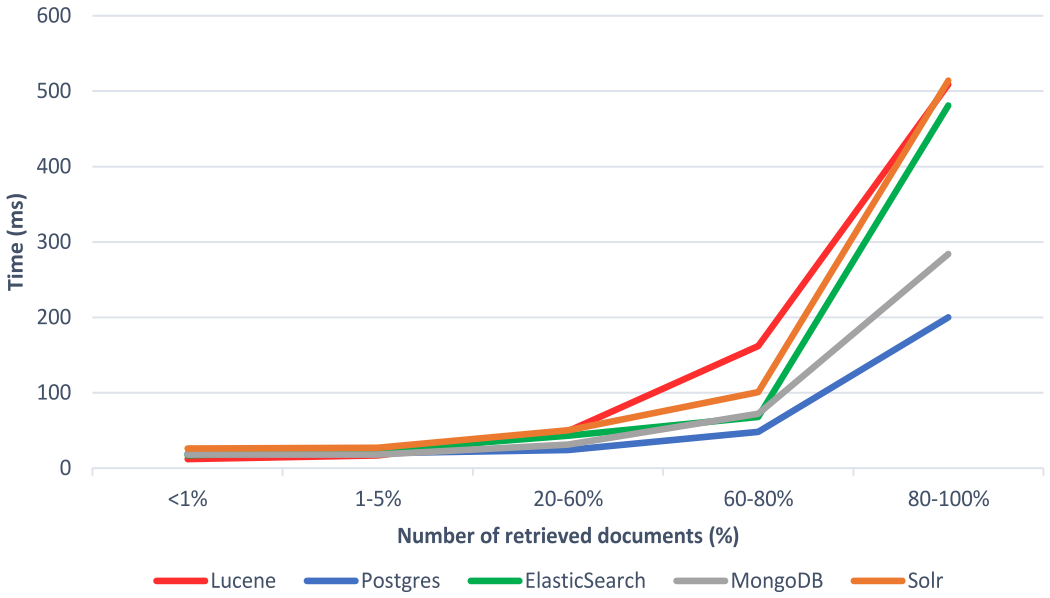

Figure 6.4: DIM Query Results on 863 DICOM Files Dataset

These files use a total of approximately 650MB disk space. Like expected, the required time to perform a given query increase proportionally with the number of retrieving results. It is possible to observe that the developed solutions got better performance in all tested queries (with some exceptions). Solr based solution had a poorer performance in the tested conditions, followed by ElasticSearch based solution, MongoDB based, and finally, the relational based solution is the one with the best results.

Table 6.4 contains the queries performed over the dataset containing approximately 150000 DICOM files.

Query	Number of Results
SOPInstanceUID:*	147859
Modality:CT	134734
PatientSex:M	57994
PatientPosition:HFS	25926
Modality:MR	6830
Modality:SC	916

Table 6.4: Queries Used over the Dataset with 147859 DICOM Files

On this dataset, the developed solutions obtained a distinguished performance while comparing to the existing one. Making an estimate based on the average of the results, Solr based solution performed 2 times faster, ElasticSearch based solution 3.24 times faster, MongoDB 3.26 times faster and with the best performance, the relational based solution 4.53 times.

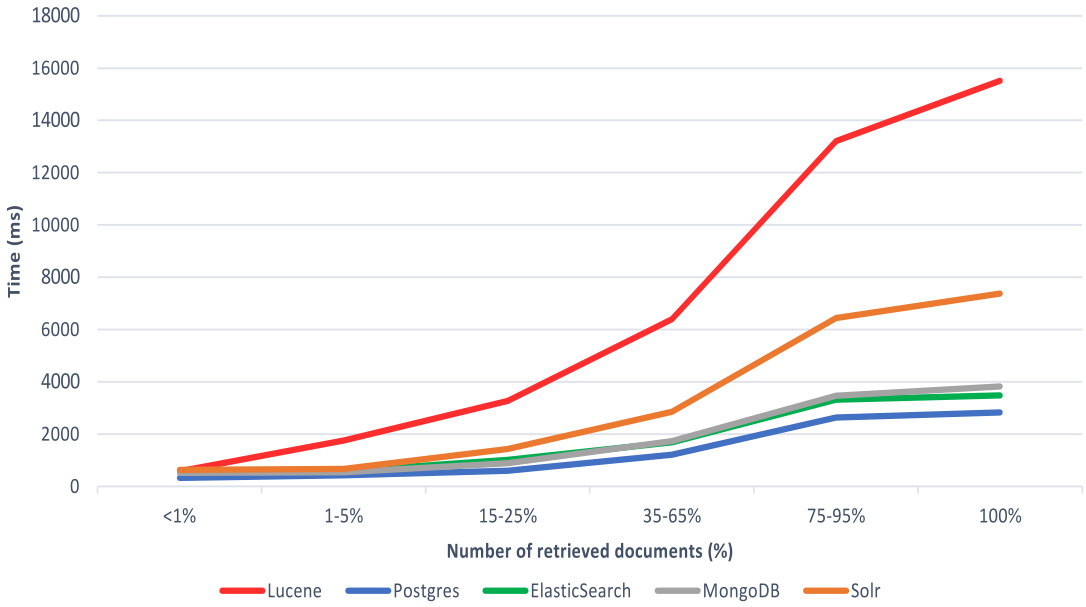

Figure 6.5: DIM Query Results on 147859 DICOM Files Dataset

On the third dataset and following the same strategy, queries on Table 6.5 were performed over a dataset with more than four million DICOM files.

Query	Number of Results
SOPInstanceUID:*	4295069
Nodality:CT	4287364
StationName:ct99	2554800
PatientSex:F	2318590
PatientAge:086Y	38135

Table 6.5: Queries Used over the Dataset with 4295069 DICOM Files

We consider that this dataset already simulates a big data scenario where the existing solutions run over because it is large and complex enough for that traditional data processing applications and some Dicoogle services became inadequate. For instance, service that saves all data in memory causes a platform crash for getting out of available memory.

It is possible to observe that all developed solutions got a better performance than the existing one, being ElasticSearch based and relational database based the ones with the best results, followed by MongoDB based and finally Solr based which as a performance close to the one obtained by the Lucene solution.

Table 6.6 contains the queries performed over the biggest dataset with more that seven million DICOM files.

Graphic 6.7 illustrated the obtain results by the different solutions on the 7524561 DICOM files dataset. The resulting files from saving query results occupied above 10GB size, with only a few fields for the document being written. ElasticSearch, MongoDB as Relational based solutions obtained quite similar performance, but MongoDB performed poorly on the wild-card query. With worst performance

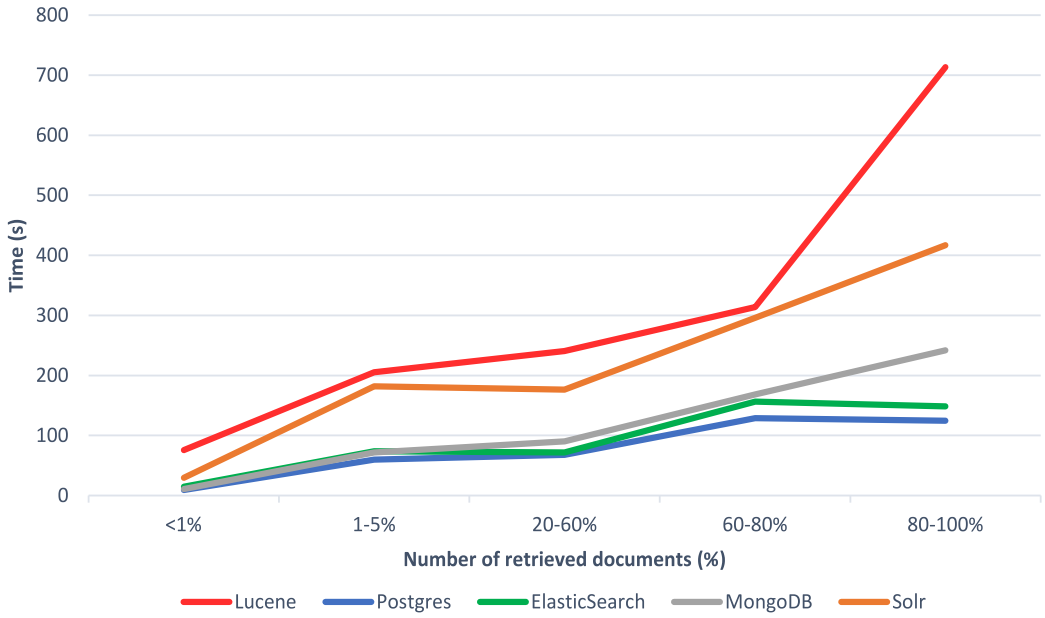

Figure 6.6: DIM Query Results on 4295069 DICOM Files Dataset

Query	Number of Results
SOPInstanceUID:*	7524561
Nodality:CT	7214799
StationName:ct99	4164050
PatientSex:M	4004726
StationName:ct01	2917562
PatientAge:077Y	210809
PatientAge:086Y	69817

Table 6.6: Queries Used over the Dataset with 7524561 DICOM Files

than these three was Solr based solution, however, it performed faster than Lucene based solution on every query.

From the above results, plenty important conclusions can be taking regarding probably the most user-concerned performance metric, the query response time. Over the smallest dataset, database based solutions have better performance in almost all queries, being the ones using inverted index techniques a little slower. The results accuracy can be questionable because small environment changes can lead to big discrepancies, also the applied caching mechanisms implemented by the engines can have a great performance impact. On larger datasets and queries with large result set size, Lucene based solution performed poorly when comparing with the remaining solutions. This can be explained mainly because of its Java implementation, that has a lower efficiency of a program executing, poor memory usage and files I/O, while database engines are written in C/C++ processed in a more efficient way[77].

Like on the index task, an outdated Lucene version also does not take full advantages of all Lucene features. On the other side, Apache Solr solves some of these questions, nonetheless, its performance is

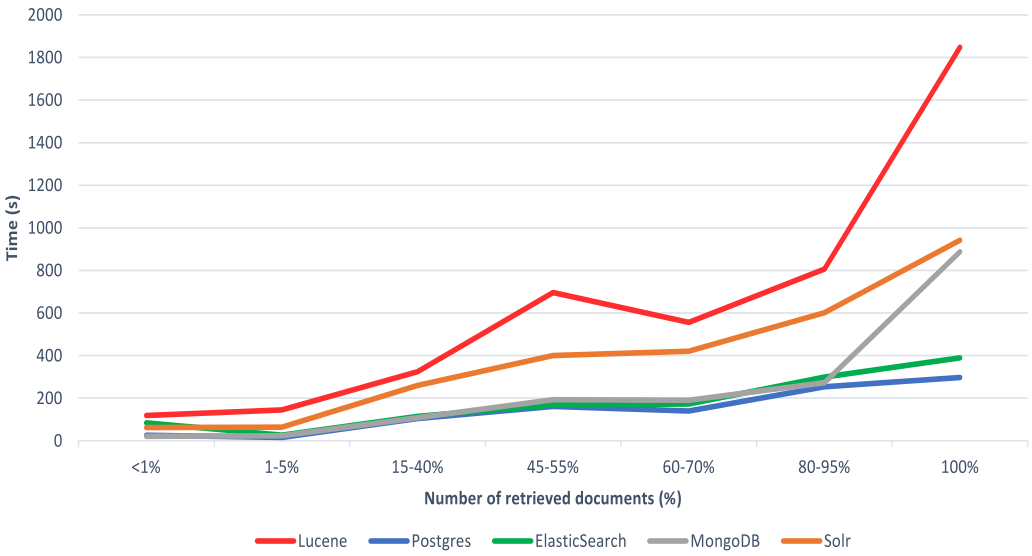

Figure 6.7: DIM Query Results on 7524561 DICOM Files Dataset

affected by the implemented retrieval strategy, because when retrieving batches of large documents it needs to sort them first based on a certain attribute, which delays the required response time.

MongoDB is probably the most powerful technology when working with a distributed scenario and leading with unstructured data, however, this is not the case. Despite the good response times, it is outperformed by ElasticSearch and PostGreSQL based Solutions. PostGreSQL engine has an optimized vertical scaling strategy, which is very useful on the proposal environment. On this DIM scenario, the data is structured following a simple hierarchical model, where the join operations from the four tables are performed in a very efficient way.

ElasticSearch has a good behavior when comparing with Apache Solr because of it scrolling, available through a Java API called `scroll api`, where large result sets can be retrieved without the need of being sorted[78].

6.4 ALL FIELDS INDEXING SCENARIO

This section contains the results from the mode of operations were the databases index all existing DICOM attributes. These solutions are quite useful for data mining/business intelligence applications, in order to explore data and meta-data. The number of fields can vary and even some vendor use specific private tags according to the different modalities. The used datasets got more than one hundred elements. Some sample tag alias that can be present in a DICOM file can be seen in Appendix 1 of this thesis.

Given the huge number of tags and values, including tags that can have nested DICOM objects, the traditional relational model cannot be applied and the implementation/comparison of performing results only was applied to the non-relational/text-based solutions.

Solutions like Solr and ElasticSearch based require that dynamic mapping existing on the latest versions was used as explained before, having a direct impact on their performance.

-Index Operation:

Every used technology has it own data structure, catching methods and other properties that influence how much time it is necessary to perform an indexing task. Like before, a graphic illustration will be made regarding the results obtained by each solution over the different datasets in the three identified performance measures.

The Graphic 6.8 illustrates the time that each solution takes in the index task over each dataset.

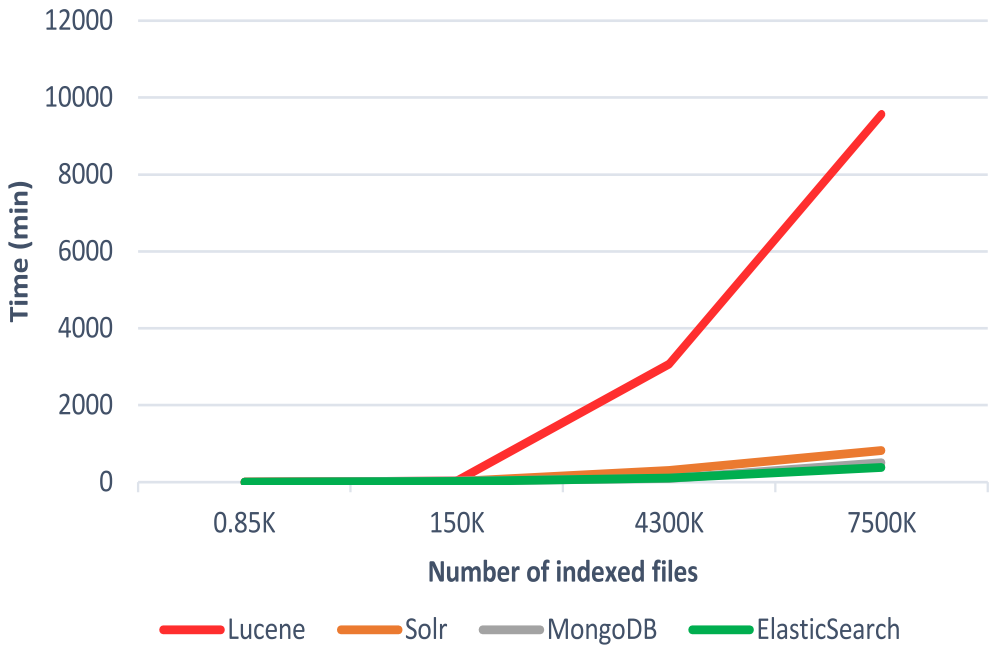

Figure 6.8: All Fields Indexing Time of all Solutions

Once again, Apache Lucene based solution demonstrated performance degradation as the datasets size increases. Figure 6.9 helps to compare the spent time between the developed solutions.

It is possible to say that ElasticSearch based solution is the one that requires less time to index all documents over the biggest datasets, while the Solr-based is slower. These index results are a quite similar to the presented on the DIM section, because although the tested datasets have more than one hundred fields per document, all engines still can handle them in an efficient way.

-Size of index on disk:

Because the number of indexed documents increases significantly, also the required space to store an inverted index/database files also increases. Figure 6.10 represents the size in megabytes used by each solution. MongoDB based solution, due to database files, starts with a greater percentage of the occupied disk when comparing with the others solutions, however, the value gets closer when the dataset size increases.

On the last dataset, all developed solutions required less disk space than the existing Lucene based, being ElasticSearch based, the one that requires less. The results can be explained following the same reasons from the DIM section, however, it is noticeable that ElasticSearch stores efficiently

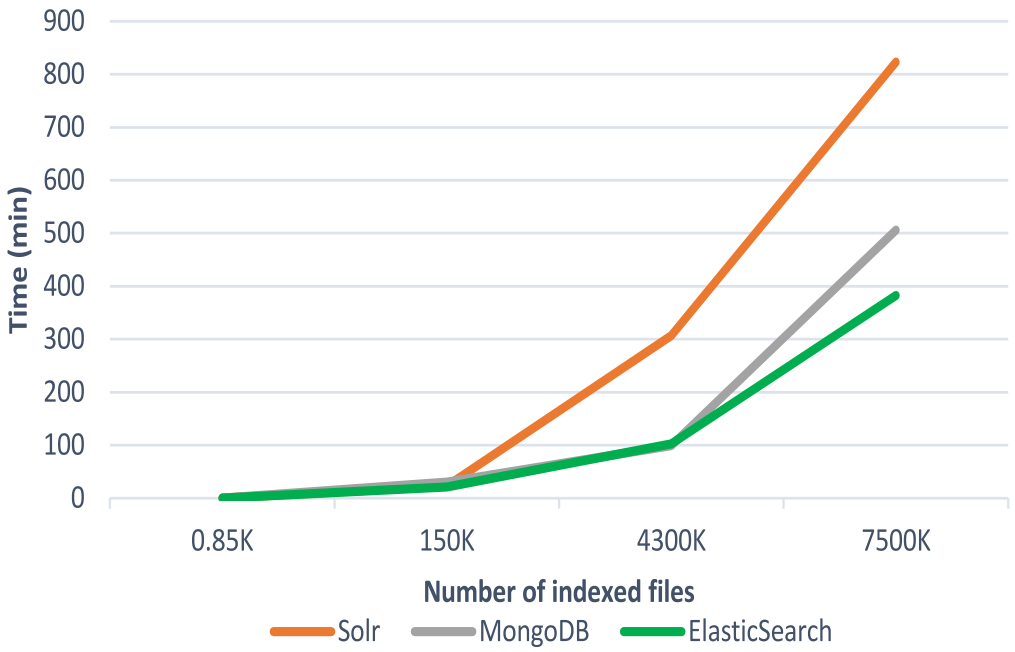

Figure 6.9: All Fields Indexing Time of the Developed Solutions

documents with a bigger number of fields[79].

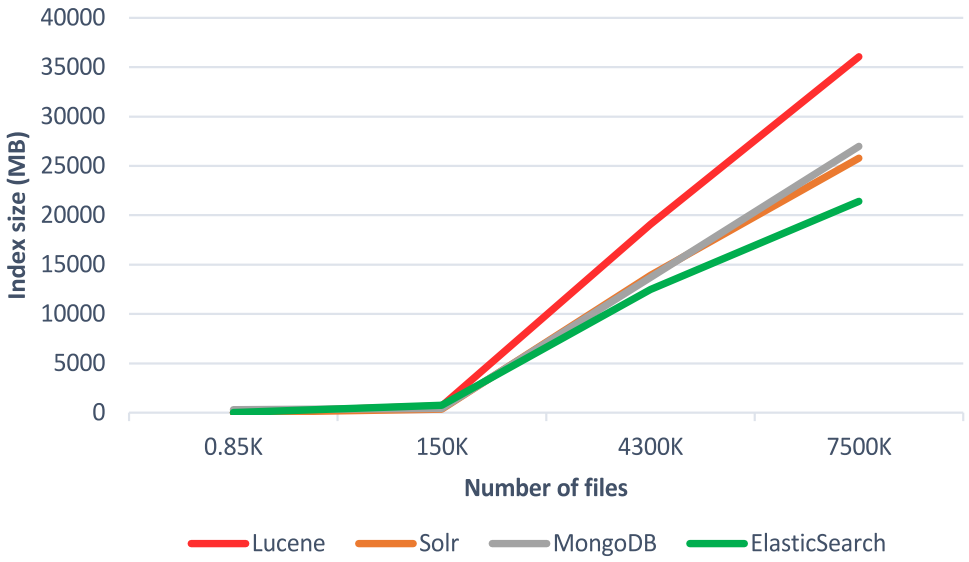

Figure 6.10: Index Size of all Solutions in the All Fields Indexing Strategy

-Query Results:

This section, like the one from the DIM results, presents the required time over the different datasets to perform several queries. The queries used are the same that the ones presented in the

referred section on the correspondent dataset because they will generate the same amount of retrieved documents.

Figure 6.11 illustrates the obtained results over the dataset with 863 DICOM files following the all fields implementation strategy. On this dataset, is not possible to take conclusions about which is the best solution because the solution with the best performance changes in different queries like expected given the small data scenario.

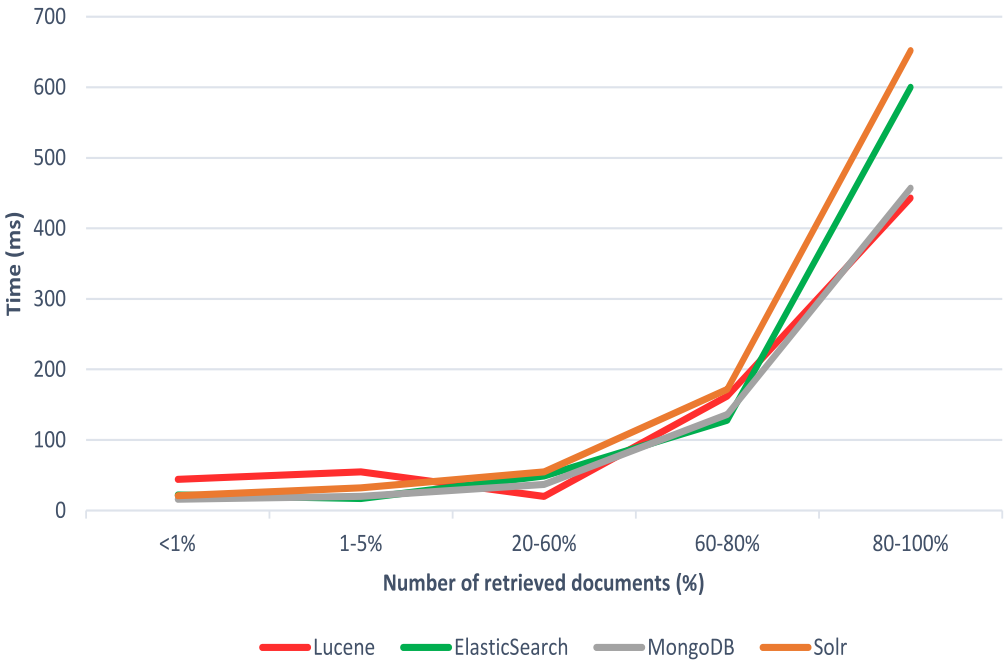

Figure 6.11: All Fields Query Results on 863 DICOM Files Dataset

Figure 6.12 represents the obtained results over the dataset with 147859 DICOM files. ElasticSearch based solutions have the best performance on all performed queries and all developed solution have better performance while comparing with the existing one, Apache Lucene based.

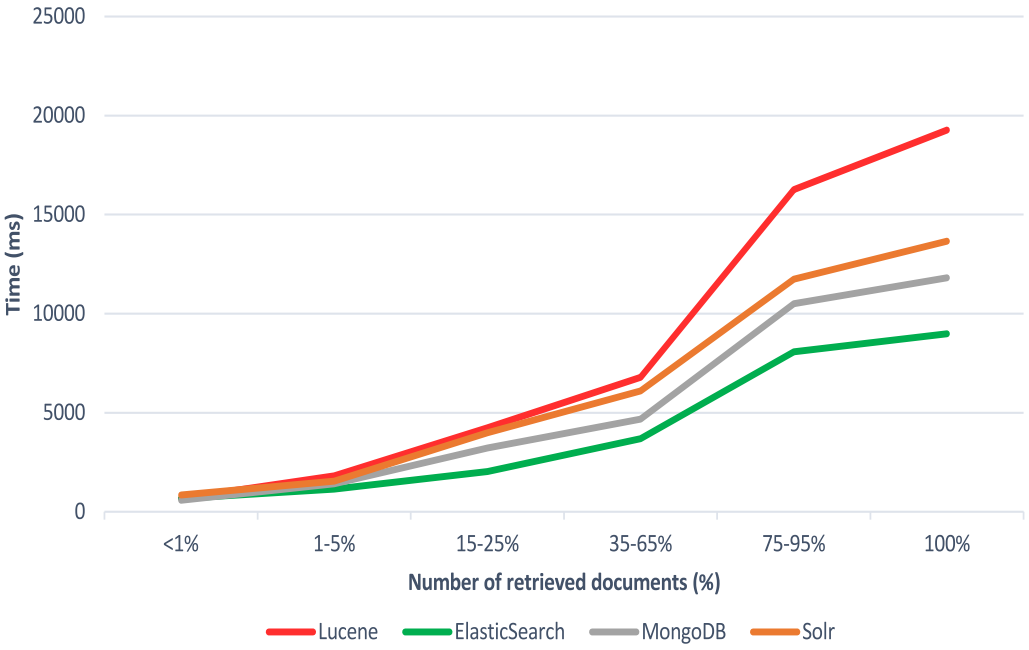

Figure 6.12: All Fields Query Results on 147859 DICOM Files Dataset

Figure 6.13 contains the obtained results over the dataset with 4295069 DICOM files. On this already considered big dataset, the performance ranking is the same as the one from the previous dataset.

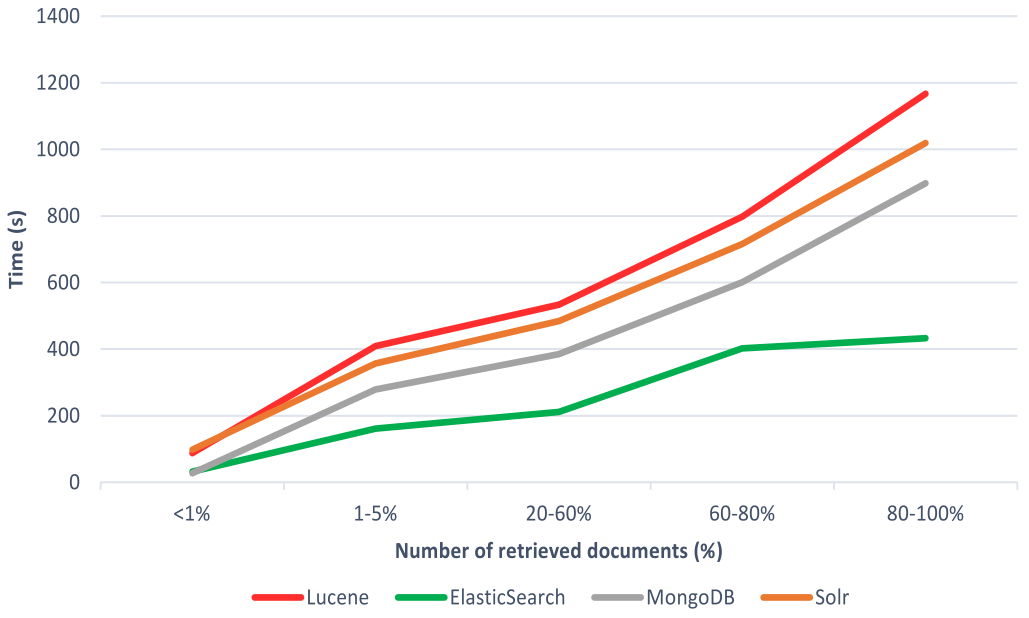

Figure 6.13: All Fields Query Results on 4295069 DICOM Files Dataset

Figure 6.14 illustrates the obtained results over the biggest dataset containing 7524561 DICOM files. ElasticSearch proved to be the fastest solution on this last dataset, performing queries notoriously faster than the Lucene solution, followed by the non-relation based solution and then by the Solr-based. The last query is a wild-card query, that MongoDB performs poorly because of its translation into a regular expression query. However, in the remaining queries, it has good performance when comparing to the remaining solutions.

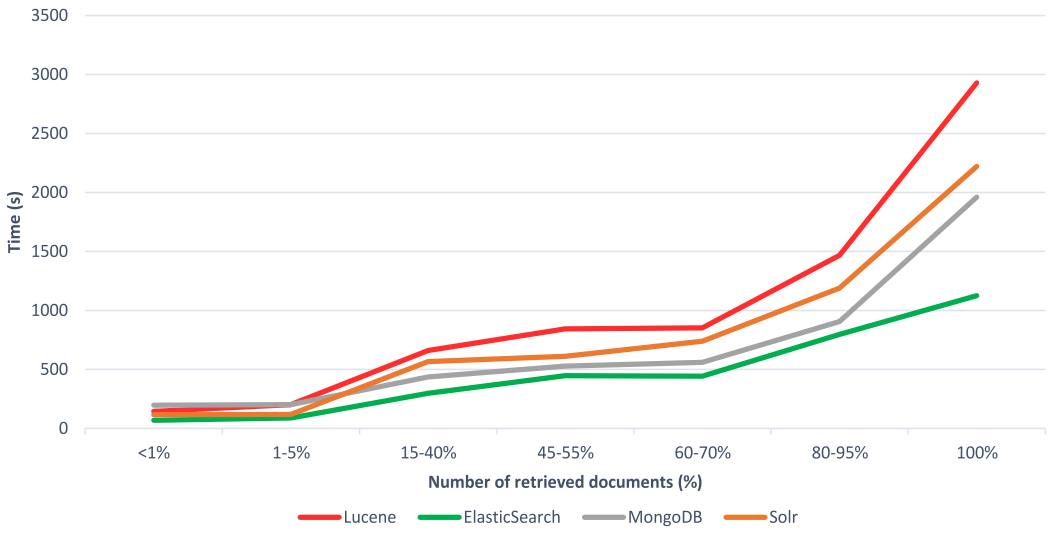

Figure 6.14: All Fields Query Results on 7524561 DICOM Files Dataset

6.5 COMPARING INDEXING STRATEGIES

After the validation of the described solutions, it is possible to conclude that all of them are suitable to replace the current Apache Lucene based solution, this because all of them got an overall better performance on the different operations. Depending on the service that will use the indexed data, a DIM or an all fields implementations should be carefully selected because they have different performance on query and index operations and required disk space also vary significantly.

Graphics 6.15 and 6.16 help to have a clearer view on the differences among the distinct data strategies. They illustrate the index and query performance on the dataset with four million DICOM files on the solutions with the best performance results that support both implementations. Regarding disk space, a DIM implementation use an average of less than half of the required space by the all fields implementation, varying with the used dataset.

On the DIM strategy, the relational based solution was the one with the best average performance in all tasks. In the all fields strategy, ElasticSearch based showed that is the best.

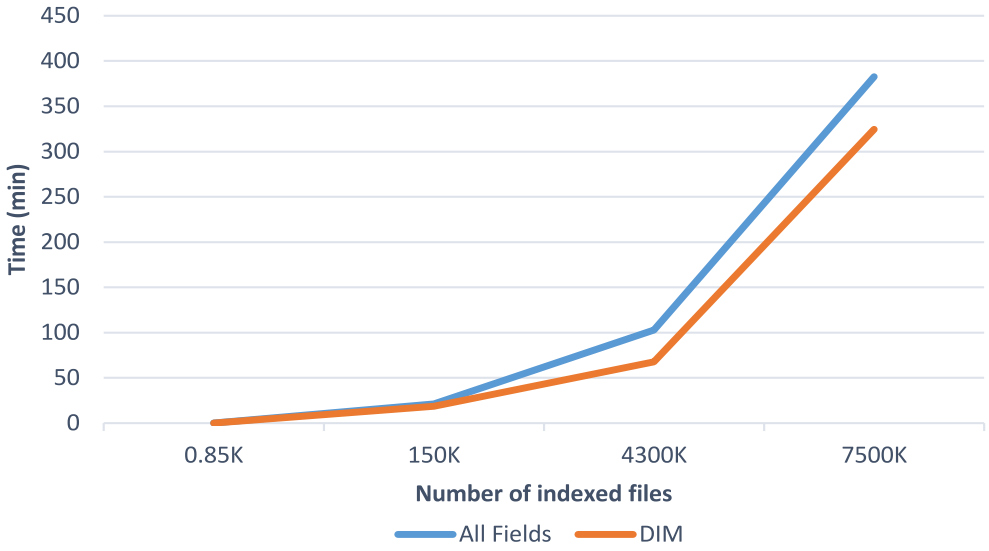

Figure 6.15: Index Results Comparing DIM and All Fields Indexing Strategies

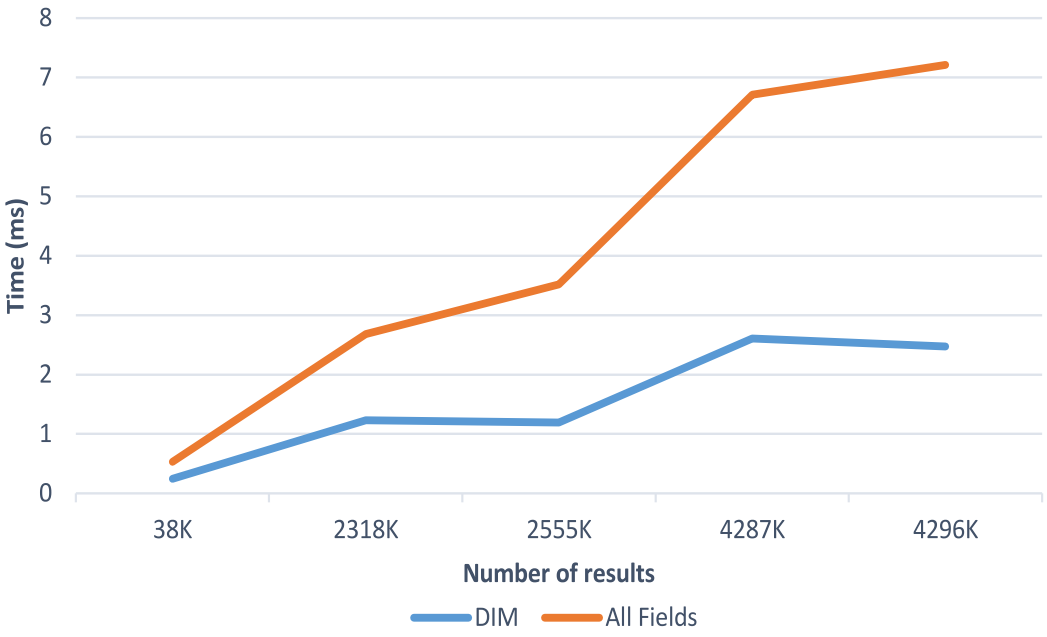

Figure 6.16: Query Results Comparing DIM and All Fields Indexing Strategies

6.6 HYBRID SOLUTION PROPOSAL

Search over all DICOM attributes to perform data analytics, data cleansing, among other big data operations and retrieve a predefined set of fields are two different services available on the Dicoogle's platform. This section proposes and describes what we consider being the database solution for the illustrated use cases, having better performance and keeping the data quality at the same level as the

current Apache Lucene based solution.

First, it will be explained the general architecture, followed by some important implementation details and finally the performance results comparing with Lucene plugin will be presented in graphical format.

6.6.1 ARCHITECTURE

Taking into account the previously illustrated results, we decide to propose a solution that has "the best of both worlds", merging the technologies with the best performance in each case: PostgreSQL and ElasticSearch based solutions.

The reasons for the selection of these two solutions is illustrated in the following topics:

PostgreSQL:

- The Advantages of this engine explained on the state of art section of this thesis.
- Best results on DIM implementation.
- Hibernate thread pool speeds up indexing task.
- Easy to set up and configure fields.

ElasticSearch:

- Best results on all fields implementation.
- Easy to set up.
- Integrity of the fields properties.
- Fast dynamic mapping ideal for the existing DICOM elements changes.

Considering that indexing always has good performance, it was decided to index all documents on both engines in order to obtain the outstanding elapsed query time on every use case. Upon receiving a query request, the solution analyses each returning field required, identifying if they all belong to the defined DIM set or not. If one or more are not indexed in the relational database, the request is sent to the ElasticSearch engine.

This verification is performed before the query translation in order to remove a possible overhead if it is an all fields query. While developing this hybrid solution, the indexed data replication on ElasticSearch (the ones indexed on PostgreSQL) was tested in order to check which has more benefits regarding query time/indexing space.

Data replication was kept although the required disk space increases. We decided to replicate the DIM fields because querying PostgreSQL to get them and query the ElasticSearch for the remaining ones in a *All Fields* query, has a huge overhead comparing to query a single engine, which can be achieved with this replication.

One implemented extra key-feature is the possibility to work with only a single engine by changing the configuration file.

Graphic 6.17 clarify the main architecture of the final developed solution.

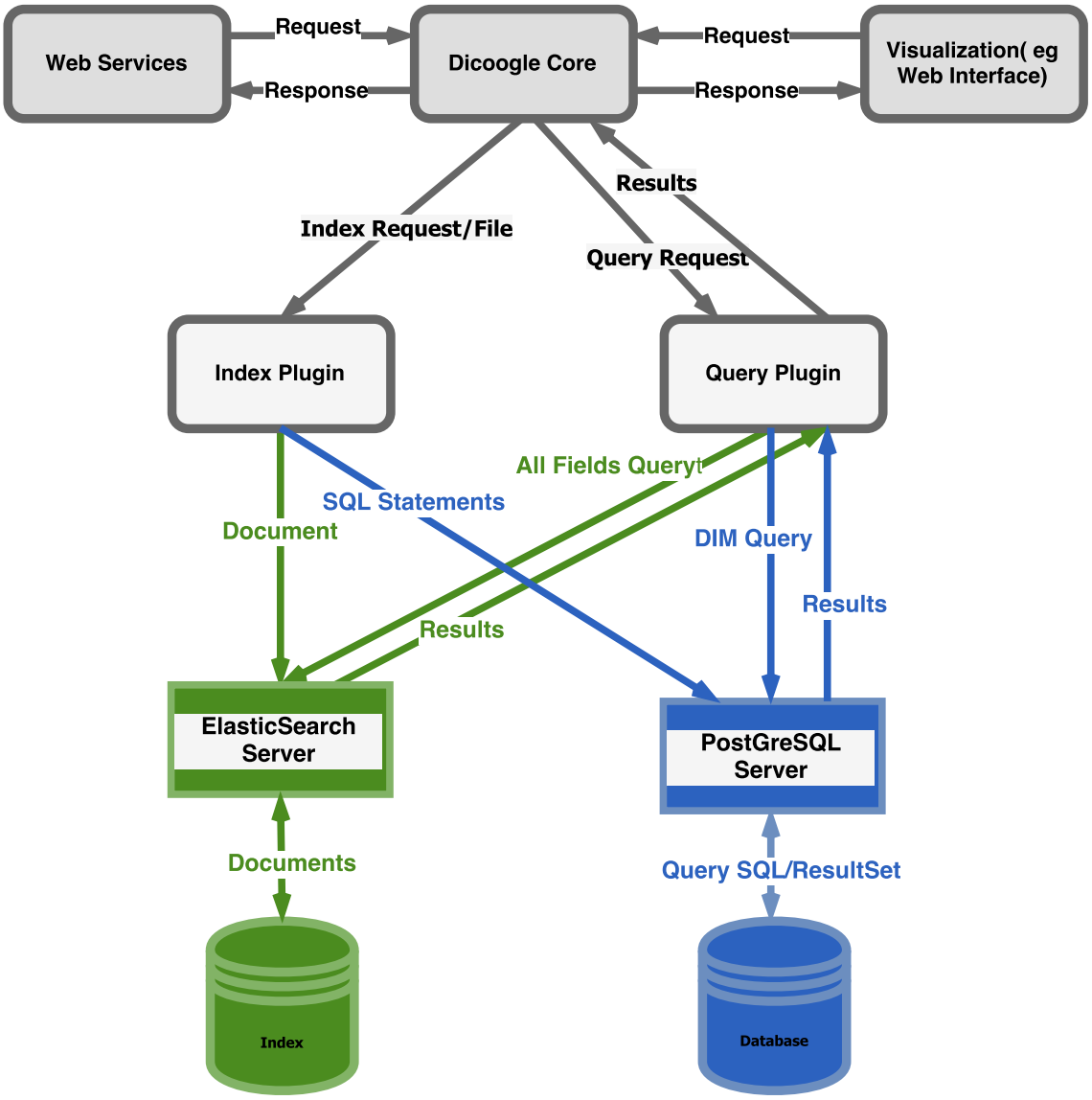

Figure 6.17: Hybrid Solution Simplified Architecture

6.6.2 PERFORMANCE EVALUATION

The next graphics compare the performance results obtained on query/index tasks on the current Lucene-based solutions against the new Hybrid solution when both engines are working, creating this way two indexes and, according to the required fields by the query, querying the correspondent one.

On graph 6.18 it is illustrated the indexing time and graphic 6.19 the results from DIM queries and all fields only over the 4 million DICOM files dataset, in order to save space and because the results are quite similar. Finally graphic 6.20 illustrates the required storage space.

We can consider an obtained performance improvement on all tested performed tests. On the smallest datasets, the hybrid solution requires more disk usage as expected due to PostgreSQL database files. However when that number of files starts scaling, they handle documents on the more efficient way, requiring even less space that the current Lucene based solution when indexing all fields. At the index task, although the index on both engines, it is visible that the hybrid solutions still outperforms both Lucene based solution index strategies. In the query operation, DIM queries over the hybrid

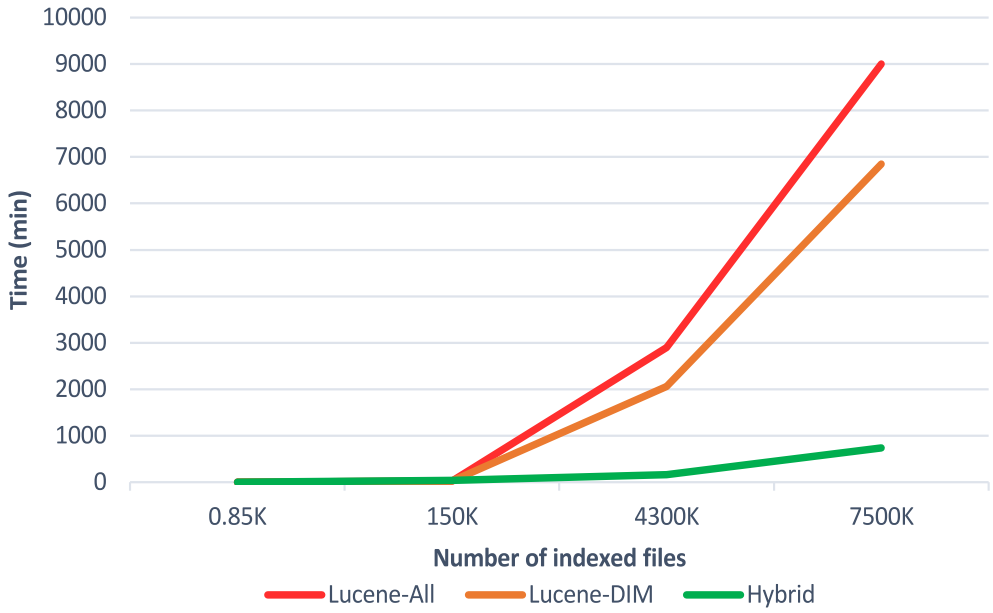

Figure 6.18: Index Operation Performance Comparison

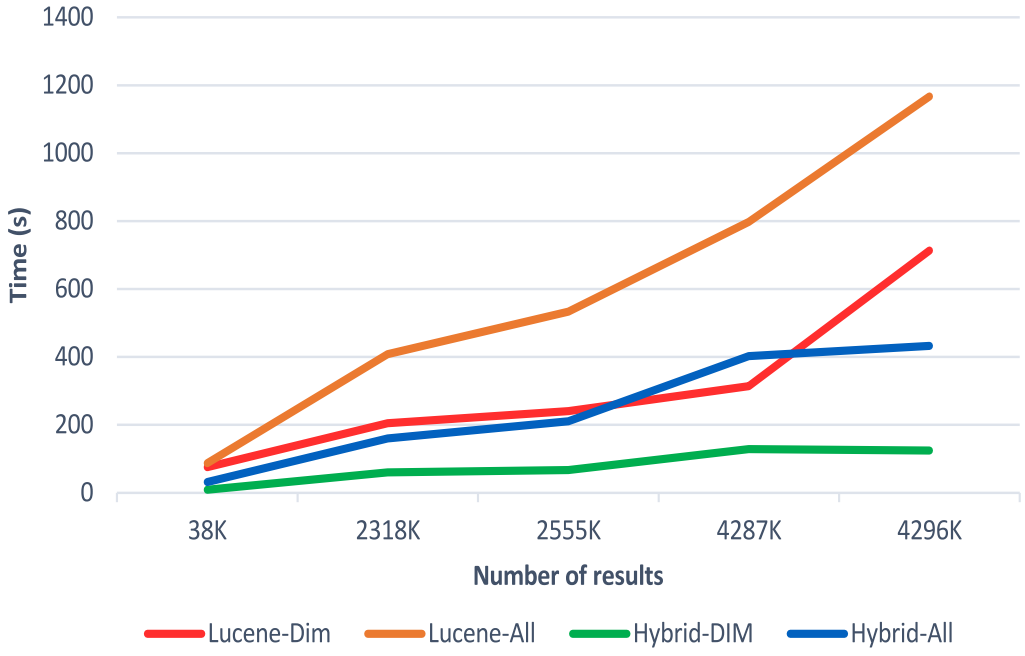

Figure 6.19: Query Operation Performance Comparison

solution are the fastest ones because they go to PostgreSQL and all fields queries are also the fastest ones because they go the ElasticSearch Engine. With this results, we think we have a final solution that can be applied on both PACS use scenarios with an acceptable performance in big data environments.

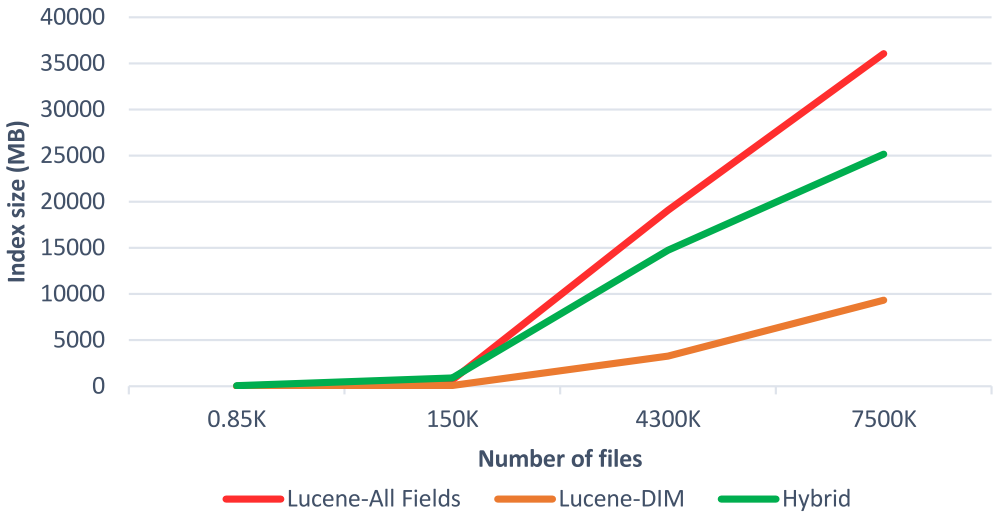

Figure 6.20: Required Disk Space Comparison

CHAPTER 7

Conclusions

This chapter summarizes and discusses the results obtained in the scope of this master thesis, as well as the experimental process that conducted to the final architectural proposal. It also provides an overview of the advantages and disadvantages of each implemented approach. Finally, it is suggested guidelines for future work and the main contributions.

7.1 Main conclusions

Nowadays, medical imaging techniques are generating enormous volumes of data which are stored in centralized repositories. With this growth, it is necessary to develop efficient methods to allow physicians and researchers to access data without breaking established workflows.

The creation of big datasets brought new challenges in the storage, search and content discovery processes. In order to satisfy these emerging requirements, indexing mechanisms and searching strategies are needed. The selection of a database technology for supporting a medical imaging repository requires a deep understanding of the real-world usage scenario and a testing framework should be in place to help the selection of the one that brings more benefits to a given implementation. This choice includes the data model (relation, non-relational or even text based) and, within each category, decide among the numerous alternatives.

Dicoogle is an open source PACS archive that replaced the traditional relational database by an indexing engine and provides enhanced data recovery functionalities, i.e. a DICOM data mining tool. Currently, Dicoogle uses the Lucene engine that has notorious scaling issues in indexing/query operations. When working with big datasets, a significant performance degradation is perceptible. Besides, the solution horizontal scaling is not possible, which limits its maximum performance to the running server specifications.

The main goal of this thesis was to develop a database architecture for replacing the current Lucene based implementation in Dicoogle. The experimental process was based on a detailed comparison of the performance between several implemented solutions using distinct technologic approaches. They

were tested executing different types of tasks and using two distinct indexing strategies, targeting the regular usage scenario that requests a minimal set of mandatory attributes and also data analytics processes that request a full indexation of DICOM attributes. Moreover, it was also investigated how the dataset size can influence the performance, mainly in big data environments. Through the experimental results, we assess the advantages and disadvantages of each approach in the programmed scenarios. As a result, a hybrid architecture is proposed and evaluated.

At the end of this thesis, four different database solutions were fully developed and tested to ensure their correctness. We can say that all of them are replacing candidates for the current Apache Lucene based solution because they obtained a notorious performance improvement over different scenarios, including the critical big data. Moreover, all of them bring value due to the running engines, including replication strategies, external compatibility frameworks, among others important features explained in this document.

In the use regular PACS usage scenario, a relational database solution got the best performing results, while the ElasticSearch approach is better on the all fields indexing. In order to obtain "the best of both worlds" a final hybrid solution is proposed.

Although Solr based solution and MongoDB based are not present in the proposed architecture, the experiments demonstrated that they have several advantages that can be useful in future scenarios. For instance, Apache Solr has an improved ranking method and MongoDB has good replication strategies.

7.2 FUTURE WORK

This work could evolve in order to improve the Dicoogle PACS index/query plugins but also to study the performance of several technologies over distinct big data environments. At the end of this thesis implementation/writing, have been identified several scenarios that may come to be implemented in the future and they will be presented on this section, announced as future work.

- Distributed scenarios can bring value to the currently developed solutions. All of the used engines got replications strategies, where sharded clusters allow to partition a dataset among a cluster of databases/full-text search instances in a way almost transparent to the application. These replications allow significant speed up on index/query tasks and it will be interesting to study them.
- The developed solutions could use multi-threading on tasks that are currently performed with a single thread strategy. Their implementations and study of the perform evolution can be seen as future work.
- Different document structures can be tested, including nested against non-nested data models to check how the solutions react to such changes.

7.3 CONTRIBUTIONS

- Improvements in Dicoogle's Platform (https://github.com/bioinformatics-ua/dicoogle)
- Alves, P., Godinho, T. M., & Costa, C. Assessing the Relational Database Model for Optimization of Content Discovery Services in Medical Imaging Repositories. In 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (accepted)

References

[1] J. Partala, N. Keraneny, M. Sarestoniemi, M. Hamalainen, J. Iinatti, T. Jamsa, J. Reponen, and T. Seppanen, "Security threats against the transmission chain of a medical health monitoring system", in *2013 IEEE 15th International Conference on e-Health Networking, Applications and Services (Healthcom 2013)*, IEEE, Oct. 2013, pp. 243–248, ISBN: 978-1-4673-5801-9. DOI: 10.1109/HealthCom.2013.6720675. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6720675.

[2] E. J. Melicio Monteiro, C. Costa, and J. L. Oliveira, "A dicom viewer based on web technology", in *2013 IEEE 15th International Conference on e-Health Networking, Applications and Services (Healthcom 2013)*, IEEE, Oct. 2013, pp. 167–171, ISBN: 978-1-4673-5801-9. DOI: 10.1109/HealthCom.2013.6720660. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6720660.

[3] E. Masai and M. Ichihashi, "Factors affecting performance of pacs.", *The Kobe journal of medical sciences*, vol. 42, no. 2, pp. 143–50, Apr. 1996, ISSN: 0023-2513. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/8699785.

[4] L. A. Bastião Silva, C. Costa, and J. L. Oliveira, "A pacs archive architecture supported on cloud services", *International Journal of Computer Assisted Radiology and Surgery*, vol. 7, no. 3, pp. 349–358, 2012.

[5] M. Rohaya, "Medical imaging trends and implementation: issues and challenges for developing countries", *Journal of Health Informatics in Developing Countries*, vol. 5, no. 1, pp. 89–98, 2011.

[6] V. N. Nekrassoff, "Mcgraw hill dictionary of scientific and technical terms", *Meta*, vol. 20, no. 2, pp. 159–162, 1975, ISSN: 0026-0452.

[7] P. Groves, B. Kayyali, D. Knott, and S. Van Kuiken, "The "big data" revolution in healthcare: accelerating value and innovation", Tech. Rep. January, 2013, pp. 1–22. [Online]. Available: http://www.images-et-reseaux.com/sites/default/files/medias/blog/2013/12/mckinsey%7B%5C_%7D131204%7B%5C_%7D-%7B%5C_%7Dthe%7B%5C_%7Dbig%7B%5C_%7Ddata%7B%5C_%7Drevolution%7B%5C_%7Din%7B%5C_%7Dhealthcare.pdf.

[8] W. Raghupathi, V. Raghupathi, and e. a. Raghupathi, "Big data analytics in healthcare: promise and potential", *Health Information Science and Systems*, vol. 2, no. 1, p. 3, 2014, ISSN: 2047-2501. DOI: 10.1186/2047-2501-2-3. [Online]. Available: http://hissjournal.biomedcentral.com/articles/10.1186/2047-2501-2-3.

[9] T. Kulhánek and M. Šárek, "Processing of medical images in virtual distributed environment", in *Proceedings of the 2009 Euro American Conference on Telematics and Information Systems New Opportunities to increase Digital Citizenship - EATIS '09*, New York, New York, USA: ACM Press, 2009, pp. 1–3, ISBN: 9781605583983. DOI: 10.1145/1551722.1551732. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1551722.1551732.

[10] H. Huang, "Enterprise pacs and image distribution", *Computerized Medical Imaging and Graphics*, vol. 27, no. 2, pp. 241–253, 2003, ISSN: 08956111. DOI: 10.1016/S0895-6111(02)00078-2.

[11] T. Godinho, "Computer methods for performance optimization in medical imaging networks", 2013.

[12] "Digital medical image fundamentals", in *PACS and Imaging Informatics*, Hoboken, NJ, USA: John Wiley & Sons, Inc., pp. 31–61. DOI: 10.1002/9780470560525.ch2. [Online]. Available: http://doi.wiley.com/10.1002/9780470560525.ch2.

[13] J. Zhang, J. Sun, and J. N. Stahl, *Pacs and web-based image distribution and display*, 2003.

[14] B. I. Reiner, D. Salkever, E. L. Siegel, F. J. Hooper, K. M. Siddiqui, and A. Musk, "Multi-institutional analysis of computed and direct radiography: part ii. economic analysis", *Radiology*, vol. 236, no. 2, pp. 420–426, 2005. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15972336.

[15] S. J. Doran, J. D'Arcy, D. J. Collins, R. Andriantsimiavona, M. Orton, D.-M. Koh, and M. O. Leach, "Informatics in radiology: development of a research pacs for analysis of functional imaging data in clinical research and clinical trials.", *Radiographics : A review publication of the Radiological Society of North America, Inc*, vol. 32, no. 7, pp. 2135–50, 2012, ISSN: 1527-1323. DOI: 10.1148/rg.327115138. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/22929148.

[16] Nema, "Dicom brochure", p. 2, 1993.

[17] O. S. Pianykh, *Digital Imaging and Communications in Medicine (DICOM)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN: 978-3-642-10849-5. DOI: 10.1007/978-3-642-10850-1. [Online]. Available: http://link.springer.com/10.1007/978-3-642-10850-1.

[18] B. Kristianto, C. Wen-yaw, and T. Yuh-show, "Dicom waveform generator", 2008.

[19] Nema, "Dicom ps3.3 2015c - information object definitions", 2015.

[20] ——, "Dicom ps3.4 2015c - service class specificationss", 2015.

[21] A. C. R. Nema, "Digital imaging and communications in medicine (dicom) part 3: Information object definitions", 2015.

[22] ——, "Digital imaging and communications in medicine (dicom) part 8: Network communication support for message exchange", 2015.

[23] ——, "Digital imaging and communications in medicine (dicom) part 7: - message exchange", 2015.

[24] ——, "Digital imaging and communications in medicine (dicom) part 18: - web access to dicom persistent objects (wado)", 2015.

[25] G. V. Koutelakis and D. K. Lymperopoulos, "Pacs through web compatible with dicom standard and wado service: advantages and implementation", *Annual International Conference of the IEEE Engineering in Medicine and Biology - Proceedings*, pp. 2601–2605, 2006.

[26] L. A. B. Silva, C. Costa, and J. L. Oliveira, "Semantic search over dicom repositories", *Proceedings - 2014 IEEE International Conference on Healthcare Informatics, ICHI 2014*, pp. 238–246, 2014. DOI: 10.1109/ICHI.2014.41.

[27] E. F. Codd, "Relational database: a practical foundation for productivity", *Communications of the ACM*, vol. 25, no. 2, pp. 109–117, Feb. 1982, ISSN: 00010782. DOI: 10.1145/358396.358400. [Online]. Available: http://portal.acm.org/citation.cfm?doid=358396.358400.

[28] F. Liu, C. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases", in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, New York, New York, USA: ACM Press, 2006, p. 563, ISBN: 1595934340.

[29] M. M. Mantei and R. G. G. Cattell, "A study of an entity-based database user interface", *ACM SIGCHI Bulletin*, vol. 14, no. 1, pp. 5–16, Jul. 1982, ISSN: 07366906. DOI: 10.1145/1044183.1044184. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1044183.1044184.

[30] E. F. Cood, *A relational model of data for large shared data banks*, 1969. [Online]. Available: http://www.seas.upenn.edu/%7B~%7Dzives/03f/cis550/codd.pdf (visited on 05/19/2016).

[31] N. Leavitt, "Will nosql databases live up to their promise?", *Computer*, vol. 43, no. 2, pp. 12–14, 2010.

[32] *Db-engines ranking*, http://db-engines.com/en/ranking, Accessed: 2016-01-27.

[33] S. Suehring, *MySQL^{TM} Bible*. Wiley Pub, 202, p. 661, ISBN: 0-7645-4932-4.

[34] M. Kofler, T. By, D. Kramer, S. Anglin, D. Appleman, E. Buckingham, G. Cornell, T. Davis, J. Gilmore, J. Hassell, C. Mills, D. Shakeshaft, and J. Sumser, "The definitive guide to mysql5 the definitive guide to mysql 5 contents at a glance", [Online]. Available: http://www.springeronline.com.%20http://www.apress.com..

[35] U. Dar, U. Uthup, D. Kapadia, and A. Saldanha, *PostgreSQL server programming : extend postgreSQL using postgreSQL server programming to create, test, debug, and optimize a range of user-defined functions in your favorite programming language*, p. 320, ISBN: 9781783980598.

[36] T. Conrad, "Postgresql vs. mysql vs. commercial databases: it's all about what you need", *Jupitermedia Corporation*, pp. 1–5, 2004. [Online]. Available: http://www.devx.com/dbzone/Article/20743.

[37] R. M. Lerner, "Open-source databases, part ii: postgresql", DOI: 10.1145/1250000/12. [Online]. Available: http://0-delivery.acm.org.innopac.lib.ryerson.ca/10.1145/1250000/12...

[38] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O. Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive", *SIGMOD '14 - Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1235–1246, 2014, ISSN: 07308078. DOI: 10.1145/2588555.2595630.

[39] "Profiling and analyzing the i/o performance of nosql dbs", *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 389–390, 2013, ISSN: 01635999. [Online]. Available: http://dl.acm.org/citation.cfm?id=2494232.2479782.

[40] Martin Fowler, *NoSQL Distilled*, 1. 2014, pp. 1–5. DOI: 10.1007/s13398-014-0173-7.2. eprint: arXiv:1011.1669v3.

[41] J. Jing Han, H. Haihong E, G. Guan Le, and J. Jian Du, "Survey on nosql database", in *2011 6th International Conference on Pervasive Computing and Applications*, IEEE, Oct. 2011, pp. 363–366, ISBN: 978-1-4577-0208-2. DOI: 10.1109/ICPCA.2011.6106531. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6106531.

[42] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello, "Sindice.com: a document-oriented lookup index for open linked data", *International Journal of Metadata, Semantics and Ontologies*, vol. 3, no. 1, p. 37, 2008, ISSN: 1744-2621. DOI: 10.1504/IJMSO.2008.021204. [Online]. Available: http://www.inderscience.com/link.php?id=21204.

[43] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems", 2009.

[44] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: mining maximal frequent subgraphs from graph databases", *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, no. 1, pp. 581–586, 2004. DOI: 10.1145/1014052.1014123. [Online]. Available: http://portal.acm.org/citation.cfm?id=1014052.1014123.

[45] R. Cattell, "Scalable sql and nosql data stores", *ACM SIGMOD Record*, vol. 39, no. 4, p. 12, 2011, ISSN: 01635808. DOI: 10.1145/1978915.1978919.

[46] C. Birgen Supervisors and H. Preisig John Morud, "Sql vs. nosql", 2014.

[47] K. Chodorow, *MongoDB : the definitive guide*, p. 409, ISBN: 9781449344689.

[48] "Nosql databases: mongodb vs cassandra", *Proceedings of the International C* Conference on Computer Science and Software Engineering, ACM 2013*, pp. 14–22, 2013. DOI: 10.1145/2494444.2494447.

[49] *Cassandra: The Definitive Guide*. O'Reilly, 2010, vol. 12, p. 332, ISBN: 144939664X. DOI: 10.1525/jer.2010.5.4.toc. [Online]. Available: https://books.google.com/books?hl=en%7B%5C&%7Dlr=%7B%5C&%7Did=MKGSbCbEdg0C%7B%5C&%7Dpgis=1.

[50] "Survey on nosql database", *Proceedings - 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011*, pp. 363–366, 2011, ISSN: 978-1-4577-0207-5. DOI: 10.1109/ICPCA.2011.6106531.

[51] M. Paksula, "Persisting objects in redis key-value database", *Science*, p. 6, 2010.

[52] Datastax, "Benchmarking top nosql databases", no. February, p. 11, 2013. [Online]. Available: http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf.

[53] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems a performance evaluation for open source bigtable implementations", *Evaluation*, p. 8, 2006.

[54] A. Hegde and P. Shraff, "Performance analysis of lucene index on hbase environment", pp. 1–15,

[55] B. M and B. M., *Understanding search engines: Mathematical model- ing and text retrieval*. SIAM, Society for Industrial and Applied Mathematics, 2005, p. 136.

[56] L. Hirsch, R. Hirsch, and M. Saeedi, "Evolving lucene search queries for text classification", pp. 1604–1611, 2007. [Online]. Available: http://discovery.ucl.ac.uk/141286/.

[57] Apache. (2016). Apache sf: Lucene index server.

[58] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination, "Apache lucene 4", 2012.

[59] D. Cutting and J. Pedersen, "Optimization for dynamic inverted index maintenance", *Proceedings of the 13th annual international ...*, pp. 405–411, 1989. DOI: 10.1145/96749.98245. [Online]. Available: http://doi.acm.org/10.1145/96749.98245$%5Cbackslash$nhttp://dl.acm.org/citation.cfm?id=98245.

[60] T. Rademakers, *Lucene In Action*. Manning Publications, 2010, p. 528, ISBN: 9781617290121.

[61] D. Smiley and E. Pugh, *Apache Solr 3 Enterprise Search Server*. 2009, pp. 7–10, ISBN: 9781849516068.

[62] R. Kuc and M. Rogozinski, *Elasticsearch Server*, 9. Packt Publishing, 2013, vol. 25, p. 428, ISBN: 9781849518444. DOI: 10.1017/CBO9781107415324.004. eprint: arXiv:1011.1669v3.

[63] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, "Sphinx-4 : a flexible open source framework for speech recognition", *Smli*, no. TR-2004-139, pp. 1–9, 2004. DOI: 10.1.1.91.4704.

[64] R. C. Miller and K. Bharat, "Sphinx: a framework for creating personal, site-specific web crawlers", *Computer Networks and ISDN Systems*, vol. 30, pp. 119–130, 1998.

[65] S. G. Langer, "Challenges for data storage in medical imaging research.", *Journal of digital imaging*, vol. 24, no. 2, pp. 203–7, Apr. 2011.

[66] T. M. Godinho, C. Viana-Ferreira, L. A. Bastião Silva, and C. Costa, "A routing mechanism for cloud outsourcing of medical imaging repositories.", *IEEE journal of biomedical and health informatics*, vol. 20, no. 1, pp. 367–75, Jan. 2016, ISSN: 2168-2208. DOI: 10.1109/JBHI.2014.2361633.

[67] *Enhanced regional network for medical imaging repositories*, 2013.

[68] T. M. Godinho, L. M. Silva, and C. Costa, "An automation framework for pacs workflows optimization in shared environments", in *2015 10th Iberian Conference on Information Systems and Technologies (CISTI)*, IEEE, Jun. 2015, pp. 1–7.

[69] A. Savaris, T. Härder, and A. von Wangenheim, "Dcmdsm: a dicom decomposed storage model.", *Journal of the American Medical Informatics Association : JAMIA*, vol. 21, no. 5, pp. 917–24, 2014.

[70] F. Valente, L. A. B. Silva, T. M. Godinho, and C. Costa, "Anatomy of an extensible open source pacs", *Journal of Digital Imaging*, Oct. 2015, ISSN: 0897-1889. DOI: 10.1007/s10278-015-9834-0. [Online]. Available: http://link.springer.com/10.1007/s10278-015-9834-0.

[71] C. Costa, C. Ferreira, L. Bastião, L. Ribeiro, A. Silva, and J. L. Oliveira, "Dicoogle - an open source peer-to-peer pacs.", *Journal of digital imaging*, vol. 24, no. 5, pp. 848–56, Oct. 2011. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/20981467%20http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3180530.

[72] C. Costa, F. Freitas, M. Pereira, A. Silva, and J. L. Oliveira, "Indexing and retrieving dicom data in disperse and unstructured archives.", *International journal of computer assisted radiology and surgery*, vol. 4, no. 1, pp. 71–7, Jan. 2009, ISSN: 1861-6429. DOI: 10.1007/s11548-008-0269-7. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/20033604.

[73] C. I, B. O, Richter, and S. T, "Postgresql: The suitable dbms solution for astronomy and astrophysics", *Astronomical Data Analysis Software and Systems (ADASS) XIII*, vol. 4, 2004.

[74] M. Fotache and D. Cogean, "Nosql and sql databases for mobile applications. case study: mongodb versus postgresql", *Informatica Economică*, vol. 17, no. 2, 2013.

[75] G. Veer, P. Rathod, P. Sinare, and R. Singh, "Building inverted index and search engine using apache lucene", *Ijarcce.Com*, vol. 4, no. 2, pp. 426–428, 2015. DOI: 10.17148/IJARCCE.2015.4296. [Online]. Available: http://ijarcce.com/upload/2015/february-15/IJARCCE7F.pdf.

[76] "An empirical study on performance comparison of lucene and relational database", *2009 International Conference on Communication Software and Networks*, 2009.

[77] A. Rafique, L. Albertsson, E. Zeitler, and L. Kaati, "Evaluating nosql technologies for historical financial data", 2013.

[78] X. Gao, L. Hall, S. W. Ave, J. Qiu, L. Hall, and S. W. Ave, "Scalable inverted indexing on nosql table storage", 2010.

[79] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases", *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings*, no. August 2013, pp. 15–19, 2013, ISSN: 1555-5798. DOI: 10.1109/PACRIM.2013.6625441.

Anexo 1 - All fields strategy tags sample

OverlayRows	Columns
NumberOfPhaseEncodingSteps	OverlayBitsAllocated
SeriesInstanceUID	uri
FrameOfReferenceUID	RequestedProcedureDescription
BitsAllocated	AccessionNumber
Modality	ContentTime
ImagingFrequency	PatientWeight
RequestingPhysician	ReferringPhysicianName
dBdt	NumberOfAverages
AngioFlag	SliceThickness
InPlanePhaseEncodingDirection	OverlayColumns
PatientBirthDate	SmallestImagePixelValue
OverlayBitPosition	PerformedProcedureStepStartTime
SamplesPerPixel	WindowWidth
MRAcquisitionType	DateOfLastCalibration
SAR	PerformingPhysicianName
EchoTime	SliceLocation
BitsStored	PerformedProcedureStepDescription
StudyDescription	WindowCenter
StudyInstanceUID	PerformedProcedureStepStartDate
PixelRepresentation	WindowCenterWidthExplanation
DerivationDescription	PerformedProcedureStepID
DeviceSerialNumber	LargestImagePixelValue
ScanningSequence	TimeOfLastCalibration
ProcedureCodeSequence	RescaleSlope
StationName	ImageComments
ImplementationClassUID	RescaleIntercept
ProtocolName	Manufacturer
InstanceNumber	EchoTrainLength

ContrastBolusVolume	SourceApplicationEntityTitle
SeriesDescription	SOPClassUID
ReferencedPatientSequence	PatientPosition
MediaStorageSOPClassUID	SoftwareVersions
VariableFlipAngleFlag	OverlayType
ImagedNucleus	HighBit
ContentDate	PhysiciansOfRecord
RequestAttributesSequence	OperatorName
TransmitCoilName	MagneticFieldStrength
PixelBandwidth	FlipAngle
ImplementationVersionName	StudyDate
PhotometricInterpretation	RescaleType
ReferencedImageSequence	InstanceCreationTime
MediaStorageSOPInstanceUID	SourceImageSequence
AcquisitionDate	InstanceCreationDate
RequestedProcedureID	InstitutionName
SeriesNumber	SeriesDate
ReferencedStudySequence	ManufacturerModelName
SeriesTime	PatientID
Rows	InstanceCreationDate
SequenceName	TransferSyntaxUID
RepetitionTime	PercentPhaseFieldOfView
PercentSampling	ContrastBolusAgent
AcquisitionNumber	InstitutionAddress
AcquisitionTime	RequestedCodeSequence
PatientName	StudyID
PatientAge	PatientSex

Anexo 2 - Mapping Solr DIM

```
<fieldType name="boolean" class="solr.BoolField" sortMissingLast="true"/>
<fieldType name="strings" class="solr.StrField" sortMissingLast="false" multiValued="true"/>
<fieldType name="stringss" class="solr.StrField" sortMissingLast="false" multiValued="false"/>
<fieldType name="stringsss" class="solr.StrField" sortMissingLast="false" multiValued="true"/>
<fieldType name="tlongs" class="solr.TrieLongField" positionIncrementGap="0" multiValued="false"
    precisionStep="8"/>
<fieldType name="tdoubles" class="solr.TrieDoubleField" positionIncrementGap="0" multiValued="false"
    precisionStep="8"/>
<fieldType name="string" class="solr.StrField" sortMissingLast="true"/>
<fieldType name="long" class="solr.TrieLongField" positionIncrementGap="0" precisionStep="0"/>
<fieldType name="tdate" class="solr.TrieDateField" positionIncrementGap="0" precisionStep="6"/>
<fieldType name="tdates" class="solr.TrieDateField" positionIncrementGap="0" multiValued="true"
    precisionStep="6"/>
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100" multiValued="true">

<analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
      <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
    <analyzer type="query">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
      <filter class="solr.SynonymFilterFactory" expand="true" ignoreCase="true"
          synonyms="synonyms.txt"/>
      <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
  </fieldType>

<uniqueKey>id</uniqueKey>
<field name="AccessionNumber" type="stringss" required="false"/>
<field name="InstitutionDepartmentName" type="stringss" required="false"/>
<field name="InstitutionName" type="stringss" required="false"/>
<field name="Manufacturer" type="stringss" required="false"/>
<field name="ManufacturerModelName" type="stringss" required="false"/>
<field name="Modality" type="stringss" required="false"/>
<field name="NumberOfStudyRelatedInstances" type="stringss" indexed="false" required="false"/>
<field name="OperatorName" type="stringsss" required="false"/>
```

```
<field name="PatientAge" type="stringss" required="false"/>
<field name="PatientBirthDate" type="stringss" required="false"/>
<field name="PatientID" type="stringss" required="false"/>
<field name="PatientName" type="stringss" required="false"/>
<field name="PerformedLocation" type="stringss" required="false"/>
<field name="PatientTelephoneNumbers" type="stringss" indexed="false" required="false"/>
<field name="PatientSex" type="stringss" required="false"/>
<field name="PerformingPhysicianName" type="stringss" indexed="false" required="false"/>
<field name="PregnancyStatus" type="stringss" required="false"/>
<field name="ReferringPhysicianName" type="stringss" indexed="false" required="false"/>
<field name="SOPInstanceUID" type="stringss"/>
<field name="SeriesDescription" type="stringsss" required="false"/>
<field name="SeriesInstanceUID" type="stringss" required="false"/>
<field name="SeriesNumber" type="stringss" indexed="false" required="false"/>
<field name="StudyDate" type="stringss" required="false"/>
<field name="StudyDescription" type="stringss" indexed="false" required="false"/>
<field name="StudyID" type="stringss" required="false"/>
<field name="StationName" type="stringss" required="false"/>
<field name="StudyName" type="stringss" required="false"/>
<field name="PatientPosition" type="stringss" required="false"/>
<field name="StudyInstanceUID" type="stringss" required="false"/>
<field name="StudyTime" type="stringss" required="false"/>
<field name="_root_" type="string" indexed="true" stored="false" required="false"/>
<field name="_version_" type="long" indexed="true" stored="true"/>
<field name="uri" type="stringss"/>
<field name="id" type="string" multiValued="false" indexed="true" required="true" stored="true"/>
</schema>
```

Anexo 3 - Mapping Elastic DIM

```
"dicoogle": {
"mappings": {
"filedicom": {
"properties": {
"AccessionNumber":{
 type: "string"
          },
          "InstitutionName": {
           type: "string"
          },
          "Manufacturer": {
           type: "string"
          },
          "ManufacturerModelName": {
           type: "string"
          },
          "Modality": {
           type: "string"
          },
          "OperatorName": {
            type: "string"
          },
          "PatientAge": {
            type: "string"
          },
          "PatientBirthDate": {
            type: "string"
          },
          "PatientID": {
            type: "string"
```

```
 },
"PatientName": {
   type: "string"
 },
"PatientPosition": {
   type: "string"
 },
"PatientSex": {
   type: "string"
 },
"PerformingPhysicianName": {
   type: "string"
 },
"PrivateAttribute": {
   type: "double"
 },
"ReferringPhysicianName": {
   type: "string"
 },
"SOPInstanceUID": {
   type: "string"
 },
"SeriesDescription": {
   type: "string"
 },
"SeriesInstanceUID": {
   type: "string"
 },
"SeriesNumber": {
   type: "string"
 },
"StationName": {
   type: "string"
 },
"StudyDate": {
   type: "string"
 },
"StudyDescription": {
   type: "string"
 },
"StudyID": {
   type: "string"
 },
"StudyInstanceUID": {
   type: "string"
```

```
        },
        "StudyTime": {
          type: "string"
        },
        "uri": {
          type: "string"
        }
      }
    }
  }
}
```